

UNCLASSIFIED
Technical
Report
distributed by



DEFENSE

TECHNICAL

INFORMATION

CENTER

DTIC

*Acquiring Information -
Imparting Knowledge*

DEFENSE LOGISTICS AGENCY
Cameron Station
Alexandria, Virginia 22304-6145

UNCLASSIFIED

UNCLASSIFIED

NOTICE

We are pleased to supply this document in response to your request.

The acquisition of technical reports, notes, memorandums, etc., is an active, ongoing program at the **Defense Technical Information Center (DTIC)** that depends, in part, on the efforts and interest of users and contributors.

Therefore, if you know of the existence of any significant reports, etc., that are not in the DTIC collection, we would appreciate receiving copies or information related to their sources and availability.

The appropriate regulations are Department of Defense Directive 3200.12, DoD Scientific and Technical Information Program; Department of Defense Directive 5200.20, Distribution Statements on Technical Documents (*amended by Secretary of Defense Memorandum, 18 Oct 1983, subject: Control of Unclassified Technology with Military Application*); Military Standard (MIL-STD) 847-B, Format Requirements for Scientific and Technical Reports Prepared by or for the Department of Defense; Department of Defense 5200.1R, Information Security Program Regulation.

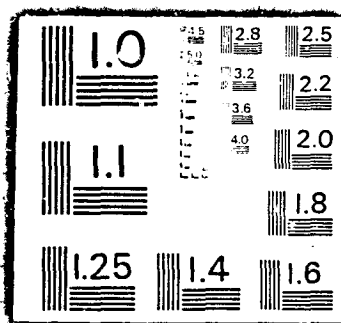
Our Acquisition Section, **DTIC-FDAB**, will assist in resolving any questions you may have. Telephone numbers of that office are: (202) 274-6847, (202) 274-6874 or Autovon 284-6847, 284-6874.

DO NOT RETURN THIS DOCUMENT TO DTIC

**EACH ACTIVITY IS RESPONSIBLE FOR DESTRUCTION OF THIS
DOCUMENT ACCORDING TO APPLICABLE REGULATIONS.**

UNCLASSIFIED

1 OF 7
AD
721 399



66312200
AD721399

MAC TR-34

PROCEDURES AS A REPRESENTATION FOR DATA IN A COMPUTER
PROGRAM FOR UNDERSTANDING NATURAL LANGUAGE

Terry Winograd

February 1971

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
PROJECT MAC

CAMBRIDGE

MASSACHUSETTS 02139

NATIONAL TECHNICAL
INFORMATION SERVICE

464

PROCEDURES AS A REPRESENTATION FOR DATA
IN A COMPUTER PROGRAM FOR UNDERSTANDING
NATURAL LANGUAGE

by

Terry Winograd
Massachusetts Institute of Technology
January 1971

Revised version of a dissertation submitted to the
Department of Mathematics on August 24, 1970 in partial
fulfillment of the requirements for the degree of Doctor of
Philosophy

ACKNOWLEDGEMENTS

The author would like to express his gratitude to the many people who helped and encouraged him in this work: to the members of the AI group at Project MAC who provided a critical forum for ideas; particularly to Carl Hewitt for introducing him to PLANNER, to Eugene Charniak and Gerald Sussman for their efforts in implementing Micro-Planner, and for many hours of provoking discussions about the problems of natural language and semantics; to Professor Seymour Papert for supervising the thesis, to Professors Marvin Minsky and Michael Fischer for their suggestions and criticisms; to the PDP-10 for patiently and uncomplainingly typing many drafts of the text; and to Carol for always.

ABSTRACT

This paper describes a system for the computer understanding of English. The system answers questions, executes commands, and accepts information in normal English dialog. It uses semantic information and context to understand discourse and to disambiguate sentences. It combines a complete syntactic analysis of each sentence with a "heuristic understander" which uses different kinds of information about a sentence, other parts of the discourse, and general information about the world in deciding what the sentence means.

It is based on the belief that a computer cannot deal reasonably with language unless it can "understand" the subject it is discussing. The program is given a detailed model of the knowledge needed by a simple robot having only a hand and an eye. We can give it instructions to manipulate toy objects, interrogate it about the scene, and give it information it will use in deduction. In addition to knowing the properties of toy objects, the program has a simple model of its own mentality. It can remember and discuss its plans and actions as well as carry them out. It enters into a dialog with a person, responding to English sentences with actions and English replies, and asking for clarification when its heuristic programs cannot understand a sentence through use of context and physical knowledge.

In the programs, syntax, semantics and inference are integrated in a "vertical" system in which each part is constantly communicating with the others. We have explored several techniques for integrating the large bodies of complex knowledge needed to understand language. We use Systemic Grammar, a type of syntactic analysis which is designed to deal with semantics. Rather than concentrating on the exact form of rules for the shapes of linguistic constituents, it is structured around choices for conveying meaning. It abstracts the relevant features of the linguistic structures which are important for interpreting their meaning.

We represent many kinds of knowledge in the form of procedures rather than tables of rules or lists of patterns. By developing special procedural languages for grammar, semantics, and deductive logic, we gain the flexibility and power of programming languages while retaining the regularity and understandability of simpler rule forms. Each piece of knowledge can be a procedure, and can call on any other piece of knowledge in the system.

Thesis Supervisor: Seymour A. Papert, Professor of Applied Mathematics

Note on the Organization of the Text

This paper was written to be readable at several different levels of detail. The Preface is intended to be understandable to a layman with no special knowledge of linguistics or computers, and gives a general idea of the purposes and methods. The Introduction gives somewhat more detail, along with a sample of a dialog with the program. It explains more specifically how the program is organized, and what theories were used in its construction.

The remaining chapters each contain a general introductory section, followed by further sections explaining the details of the programs and theories. It should be possible to get a good basic understanding of the paper by reading the introduction, followed by the first section of each chapter. In addition, there was an attempt to keep sections independent so that parts of the paper could be selected separately. For example, the description of the PRGRAMMAR language, and the description of our grammar of English can be read independently.

Because of this structure, some parts of the paper are redundant -- a particularly important principle or relevant example may be repeated at all three levels of explanation, and in the different sections where it is needed. However it is hoped that it will allow the reader to go into the content as deeply as he wants without getting bogged down in detail. Throughout the text, single-spaced paragraphs are used for emphasis.

Table of Contents

Table of Contents	5
Preface -- Talking to Computers	9
<u>Chapter 1. Introduction</u>	16
1.1 General Description	16
1.1.1 What is Language?	18
1.1.2 Organization of the Language Understanding Process	22
1.2 Implementation of the System	29
1.3 Sample Dialog	35
<u>Chapter 2. Syntax</u>	61
2.1 Basic Approach to Syntax	61
2.1.1 Syntax and Meaning	62
2.1.2 Parsing	66
2.2 A Description of PROGRAMMAR	69
2.2.1 Grammar and Computers	69
2.2.2 Context-free and Context-sensitive Grammars	72
2.2.3 Systemic Grammar	73
2.2.4 Grammars as Programs	77
2.2.5 The Form of PROGRAMMAR Grammars	79
2.2.6 Context-Sensitive Aspects	83
2.2.7 Ambiguity and Understanding	88
2.2.8 Summary	91
2.3 A Grammar of English	92
2.3.1 About the Grammar	92
2.3.2 Units, Rank, and Features	95
2.3.3 The CLAUSE	104
2.3.4 Transitivity in the Clause	115
2.3.5 Noun Groups	120
2.3.6 Preposition Groups	128
2.3.7 Adjective Groups	131
2.3.8 Verb Groups	133
2.3.9 Words	139

2.3.10	Conjunction	149
2.3.11	Following the Parser in Operation	153
2.3.12	Analysis of Word Endings	171
	2.4 Programming Details	176
2.4.1	Operation of the System	176
2.4.2	Special Words	178
2.4.3	Possessives	183
2.4.4	The Dictionary	184
2.4.5	Backup Facilities	186
2.4.6	Auxilliary Functions	189
2.4.7	Messages	190
2.4.8	The Form of the Parsing Tree	191
2.4.9	Variables Maintained by the System	192
2.4.10	Pointers	193
2.4.11	Feature Manipulating	195
	2.5 Comparison with Other Parsers	196
2.5.1	Older Parsers	196
2.5.2	Augmented Transition Networks	199
2.5.3	Networks and Programs	201
	<u>Chapter 3. Inference</u>	206
	3.1 Basic Approach to Meaning	206
3.1.1	Representing Knowledge	206
3.1.2	Philosophical Considerations	210
3.1.3	Complex Information	217
3.1.4	Questions, Statements, and Commands	219
	3.2 Comparison with Previous Programs	221
3.2.1	Special Format Systems	222
3.2.2	Text Based Systems	224
3.2.3	Limited Logic Systems	226
3.2.4	General Deductive Systems	230
3.2.5	Procedural Deductive Systems	234
	3.3 Programming in PLANNER	239
3.3.1	Basic Operation of PLANNER	239
3.3.2	Backup	244
3.3.3	Differences with Other Theorem-Provers and Languages	247
3.3.4	Controlling the Data Base	250
3.3.5	Events and States	253
3.3.6	PLANNER Functions	257

3.4 The BLOCKS World	260
3.4.1 Objects	261
3.4.2 Relations	265
3.4.3 Actions	269
3.4.4 Carrying Out Commands	271
3.4.5 Memory	276
<u>Chapter 4. Semantics</u>	280
4.1 What is Semantics?	280
4.1.1 The Province of Semantics	280
4.1.2 The Semantic System	283
4.1.3 Words	286
4.1.4 Ambiguity	290
4.1.5 Discourse	294
4.1.6 Goals of a Semantic Theory	298
4.2 Semantic Structures	300
4.2.1 Object Semantic Structures	301
4.2.2 Relative Clauses	310
4.2.3 Preposition Groups	316
4.2.4 Types of Object Descriptions	318
4.2.5 The Meaning of Questions	324
4.2.6 Interpreting Imperatives	331
4.2.7 Accepting Declarative Information	333
4.2.8 Time	337
4.2.9 Semantics of Conjunction	347
4.2.10 More on Ambiguity	350
4.2.11 To Be and To Have	356
4.2.12 Additional Semantic Information	361
4.3 The Semantics of Discourse	369
4.3.1 Pronouns	371
4.3.2 Substitutes and Incompletes	377
4.3.3 Overall Discourse Context	380
4.4 Generation of Responses	384
4.4.1 Patterned Responses	384
4.4.2 Answering Questions	387
4.4.3 Naming Objects and Events	393
4.4.4 Generating Discourse	396
4.4.5 Future Development	399

4.5 Comparison with Other Semantic Systems	401
4.5.1 Introduction	401
4.5.2 Categorization	403
4.5.3 Association	406
4.5.4 Procedure	409
4.5.5 Evaluating the Models	411
4.5.6 Conclusions	420
<u>Chapter 5. Conclusions</u>	422
5.1 Teaching, Telling and Learning	422
5.1.1 Types of Knowledge	423
5.1.2 Syntax	425
5.1.3 Inference	430
5.1.4 Semantics	435
5.2 Directions for Future Research	438
Appendix A - Index of Syntactic Features	444
Appendix B - Sample Parsings	447
Appendix C - Sample BLOCKS Theorems	450
Appendix D - Sample PROGRAMMAR Program	452
Appendix E - Sample Dictionary Entries	454
Appendix F - PLANNER Data for Dialog in Sect. 1.3	457
Bibliography	458

Preface -- Talking to Computers

Computers are being used today to take over many of our jobs. They can perform millions of calculations in a second, handle mountains of data, and perform routine office work much more efficiently and accurately than humans. But when it comes to telling them what to do, they are tyrants. They insist on being spoken to in special computer languages, and act as though they can't even understand a simple English sentence.

Let us envision a new way of using computers so they can take instructions in a way suited to their jobs. We will talk to them just as we talk to a research assistant, librarian, or secretary, and they will carry out our commands and provide us with the information we ask for. If our instructions aren't clear enough, they will ask for more information before they do what we want, and this dialog will all be in English.

Why isn't this being done now? Aren't computers translating foreign languages and conducting psychiatric interviews? Surely it must be easier to understand simple requests for information than to understand Russian or a person's psychological problems. The key to this question is in understanding what we mean by "understanding". Computers are very adept at manipulating symbols -- at shuffling around strings of letters and words, looking them up in dictionaries, and rearranging them. In the early days of computing, some people thought that simple applications of these capabilities

might be just what was needed to translate languages. The government supported a tremendous amount of research into language translation, and a number of projects tried different approaches. In 1966 a committee of the National Academy of Sciences wrote a report evaluating this research and announced sadly that it had been a failure. Every project ran up against the same brick wall -- the computer didn't know what it was talking about.

When a human reader sees a sentence, he uses knowledge to understand it. This includes not only grammar, but also his knowledge about words, the context of the sentence, and most important, his knowledge about the subject matter. A computer program supplied with only a grammar for manipulating the syntax of language could not produce a translation of reasonable quality.

Everyone has heard the story of the computer that tried to translate "The spirit is willing but the flesh is weak." into Russian and came out with something which meant "The vodka is strong but the meat is rotten." Unfortunately the problem is much more serious than just choosing the wrong words when translating idioms. It isn't always possible to even choose the right grammatical forms. We may want to translate the two sentences "A message was delivered by the next visitor." and "A message was delivered by the next day." If we are translating into a language which doesn't have the equivalent of our "passive voice", we may need to completely rearrange the first sentence into something corresponding to "The next visitor delivered a message." The other sentence might become something

like "Before the next day, someone delivered a message." If the computer picks the wrong form for either sentence, the meaning is totally garbled. In order to make the choice, it has to know that visitors are people who can deliver messages, while days are units of time and cannot. It has to "understand" the meanings of the words "day" and "visitor".

In other cases the problem is even worse. Even a knowledge of the meanings of words is not enough. Let us try to translate the two sentences:

"The city councilmen refused to give the women a permit for a demonstration because they feared violence."
and

"The city councilmen refused to give the women a permit for a demonstration because they advocated revolution."

If we are translating into a language (like French) which has different forms of the word "they" for masculine and feminine, we cannot leave the reader to figure out who "they" refers to. The computer must make a choice and if it chooses wrong, the meaning of the sentence is changed. To make the decision, it has to have more than the meanings of words. It has to have the information and reasoning power to realize that city councilmen are usually staunch advocates of law and order, but are hardly likely to be revolutionaries.

For some uses, it isn't really necessary to understand

much. There has been much publicity about a well known "psycniatrist" program named ELIZA. It imitates the kind of Rogerian psychiatrist who would respond to a question like "What time is it?" by asking "Why do you want to know what time it is?" or muttering "You want to know what time it is!". This can be done without much understanding. All it needs to do is take the words of the question and rearrange them in some simple way to make a new question or statement. In addition it recognizes a few key words, to respond with a fixed phrase whenever the patient uses one of them. If the patient types a sentence containing the word "mother", the program can say "Tell me more about your family!". In fact, this is just how the psychiatrist program works. But very often it doesn't work -- its answers are silly or meaningless because it isn't really understanding the content of what is being said.

If we really want computers to understand us, we need to give them the ability to use more knowledge. In addition to a grammar of the language, they need to have all sorts of knowledge about the subject they are discussing, and they have to use reasoning to combine facts in the right way to understand a sentence and respond to it. The process of understanding a sentence has to combine grammar, semantics, and reasoning in a very intimate way, calling on each part to help with the others.

This thesis explores one way of giving the computer knowledge in a flexible and usable form. In addition to basic tools and operations for understanding language, we give the computer specialized information about the English language, the words we will use, and the subject we will discuss. In most earlier computer programs for understanding language, there have

been attempts to use these kinds of information in the form of lists of rules, patterns, and formulas.

In our system, knowledge is expressed as programs in special languages designed for syntax, semantics, and reasoning. These languages have the control structure of a programming language, with the statements of the language explicitly controlling the process. This makes it possible to relate the different areas of knowledge more directly and completely. The course of the understanding process can be determined directly by special knowledge about a word, a syntactic construction, or a particular fact about the world.

This gives greater flexibility than a program with a fixed control structure, in which the specific knowledge can only indirectly control the process of understanding. By using languages specially developed for representing these kinds of knowledge, it is possible for a person to "teach" the computer what it needs to know about a new subject or a new vocabulary without being concerned with the details of how the computer will go about using the knowledge to understand language. For simple information, it is even possible to just "tell" the computer in English. Other systems make it possible to "tell" the computer new things by allowing it to accept only very specialized kinds of information. By representing information as programs, we can greatly expand the range of things which can be included.

The best way to experiment with such ideas is to write a working program which can actually understand language. We would like a program which can answer questions, carry out commands, and accept new information in English. If we really

much. There has been much publicity about a well known "psychiatrist" program named ELIZA. It imitates the kind of Rogerian psychiatrist who would respond to a question like "What time is it?" by asking "Why do you want to know what time it is?" or muttering "You want to know what time it is!". This can be done without much understanding. All it needs to do is take the words of the question and rearrange them in some simple way to make a new question or statement. In addition it recognizes a few key words, to respond with a fixed phrase whenever the patient uses one of them. If the patient types a sentence containing the word "mother", the program can say "Tell me more about your family!". In fact, this is just how the psychiatrist program works. But very often it doesn't work -- its answers are silly or meaningless because it isn't really understanding the content of what is being said.

If we really want computers to understand us, we need to give them the ability to use more knowledge. In addition to a grammar of the language, they need to have all sorts of knowledge about the subject they are discussing, and they have to use reasoning to combine facts in the right way to understand a sentence and respond to it. The process of understanding a sentence has to combine grammar, semantics, and reasoning in a very intimate way, calling on each part to help with the others.

This thesis explores one way of giving the computer knowledge in a flexible and usable form. In addition to basic tools and operations for understanding language, we give the computer specialized information about the English language, the words we will use, and the subject we will discuss. In most earlier computer programs for understanding language, there have

been attempts to use these kinds of information in the form of lists of rules, patterns, and formulas.

In our system, knowledge is expressed as programs in special languages designed for syntax, semantics, and reasoning. These languages have the control structure of a programming language, with the statements of the language explicitly controlling the process. This makes it possible to relate the different areas of knowledge more directly and completely. The course of the understanding process can be determined directly by special knowledge about a word, a syntactic construction, or a particular fact about the world.

This gives greater flexibility than a program with a fixed control structure, in which the specific knowledge can only indirectly control the process of understanding. By using languages specially developed for representing these kinds of knowledge, it is possible for a person to "teach" the computer what it needs to know about a new subject or a new vocabulary without being concerned with the details of how the computer will go about using the knowledge to understand language. For simple information, it is even possible to just "tell" the computer in English. Other systems make it possible to "tell" the computer new things by allowing it to accept only very specialized kinds of information. By representing information as programs, we can greatly expand the range of things which can be included.

The best way to experiment with such ideas is to write a working program which can actually understand language. We would like a program which can answer questions, carry out commands, and accept new information in English. If we really

want it to understand language, we must give it knowledge about the specific subject we want to talk about.

For our experiment, we pretended that we were talking to a simple robot, with a hand and an eye and the ability to manipulate toy blocks on a table. We can say, "Pick up a block which is bigger than the one you are holding and put it in the box.", or ask a sequence of questions like "Had you touched any pyramid before you put the green one on the little cube?" "When did you pick it up?" "Why?", or we can give it new information like "I like blocks which are not red, but I don't like anything which supports a pyramid." The "robot" responds by carrying out the commands (in a simulated scene on a display screen attached to the computer), typing out answers to the questions, and accepting the information to use in reasoning later on.

The dialog is carried out by typing on a terminal attached to the computer time-sharing system. There are a number of hard technical problems in getting a computer to communicate by voice, and it has not been attempted.

We had three main kinds of goals in writing such a program. The first is the practical goal of having a language-understanding system. Even though we used the robot as our test area, the language programs do not depend on any special subject matter, and they have been adapted to other uses.

The second goal is gaining a better understanding of what language is and how it is put together. To write a program we need to make all of our knowledge about language very explicit, and we have to be concerned with the entire language process, not just one area such as syntax. We need the most advanced theories which linguists and others have developed, and we must fit them together to get the program working. This provides a rigid test for linguistic theories, and leads us into making new

theories to fill the places where the old ones are lacking. To make a computer understand language, we have to increase our knowledge of how people understand language, and one important goal is the improvement of our comprehension of language and linguistics.

More generally, we want to understand what intelligence is and how it can be put into computers. Language is one of the most complex and unique of human activities, and understanding its structure may lead to a better theory of how our minds work. The techniques needed to write a language-understanding program may be useful in many other areas of intelligence such as vision, mathematical problem solving, and game playing. All of these areas have similar problems of integrating large amounts of knowledge into a flexible system.

With advances in artificial intelligence we will some day be able to combine all of these areas to provide computers which are not tyrants, but can understand what we tell them and do what we want them to in an intelligent way.

Chapter 1 -- Introduction

1.1 General Description

This paper describes a system for the understanding of English by a computer. The system answers questions, executes commands, and accepts information in normal English dialog. It uses semantic information and context to understand discourse and to disambiguate sentences both syntactically and semantically. It combines a complete syntactic analysis of each sentence with a "heuristic understander" which uses different kinds of information about a sentence, other parts of the discourse, and general information about the world in deciding what a sentence means.

It is based on the belief that a computer cannot deal reasonably with language unless it can "understand" the subject it is discussing. The program is given a detailed model of a small part of knowledge, and it can understand sentences concerning the subject it knows about. We use as an example the knowledge needed by a simple robot consisting of only an eye and a hand. We can give it instructions to manipulate toy objects, interrogate it about the scene, and give it information it will use in deduction.

In addition to knowing the properties of blocks, hands, etc., the program has a simple model of its own mentality. It can remember and discuss its plans and actions as well as carry

them out. It enters into a true dialog with a person, responding to English sentences with actions and English replies, and asking for clarification when its heuristic programs cannot disambiguate a sentence through use of context or physical knowledge.

1.1.1 What is Language?

To write a computer program which understands natural language, we need to understand what language is and what it does. It should be approached not as a set of mathematical rules and symbols, but as a system intended to communicate ideas from a speaker to a hearer, and we want to analyze how it achieves that communication. It can be viewed as a process of translation from a structure of "concepts" in the mind of the speaker, into a string of sounds or written marks, and back into concepts in the mind of the hearer.

In order to talk about concepts, we must understand the importance of mental models (see <Minsky 1965>). In the flood of data pouring into our brains every moment, people recognize regular and recurrent patterns. From these we set up a model of the world which serves as a framework in which to organize our thoughts. We abstract the presence of particular objects, having properties, and entering into events and relationships. Our thinking is a process of manipulating the "concepts" which make up this model. Of course, there is no way of actually observing the internal workings of a person's mind, but in Section 3.1 we will discuss the justification for postulating such a "model" in analyzing the human use of language. In Section 3.4 we show what this model might look like for a small area of knowledge, and describe how it can be used for reasoning.

When we communicate with others, we select concepts and patterns from the model and map them onto patterns of sound, which are then reinterpreted by the hearer in terms of his own model. A theory can concentrate on either half of this process of generation and interpretation of language. Even though a complete theory must account for both, its approach is strongly colored by which one it views as logically primary. Most current theories are "generative", but it seems more interesting to look at the interpretive side (see <Winograd 1969> for a discussion of the issues involved). The first task a child faces is understanding rather than producing language, and he understands many utterances before he can speak any. At every stage of development, a person can understand a much wider range of patterns than he produces (see <Miller>, Chapter 7). A program is not a detailed psychological theory of how a person interprets language, but there may in fact be very informative parallels, and at a high level, it may be a reasonable simulation.

Language understanding is a kind of intellectual activity, in which a pattern of sounds or written marks is interpreted into a structure of concepts in the mind of the interpreter. We cannot think of it as being done in simple steps: 1. Parse; 2. Understand the meaning; 3. Think about the meaning. The way we parse a sentence is controlled by a continuing semantic interpretation which guides us in a "meaningful" direction.

When we see the sentence "He gave the boy plants to water." we don't get tangled up in an interpretation which would be parallel to "He gave the house plants to charity." The phrase "boy plants" doesn't make sense like "house plants" or "boy scouts", so we reject any parsing which would use it.

Syntax, semantics, and Inference must be integrated in a close way, so that they can share in the responsibility for interpretation. Our program must incorporate the flexibility needed for this kind of "vertical" system in which each part is constantly talking to the others. We have explored several techniques for integrating the large bodies of complex knowledge needed to understand language. Two are particularly important.

First, we use a type of syntactic analysis which is designed to deal with questions of semantics. Rather than concentrating on the exact form of rules for shuffling around linguistic symbols, it studies the way language is structured around choices for conveying meaning. The parsing of a sentence indicates its detailed structure, but more important it abstracts the "features" of the linguistic components which are important for interpreting their meaning. The syntactic theory includes an analysis of the way language is structured to convey information through systematic choices of features. The other parts of the program can look directly at these relevant features, rather than having to deal with minor details of the way the parsing tree looks.

Second, we represent knowledge in the form of procedures rather than tables of rules or lists of patterns. By developing special procedural languages for grammar, semantics, and deductive logic, we gain the flexibility and power of programs while retaining the regularity and understandability of simpler rule forms. Since each piece of knowledge can be a procedure, it can call on any other piece of knowledge of any type. The parser can call semantic routines to see whether the line of parsing it is following makes any sense, and the semantic routines can call deductive programs to see whether a particular phrase makes sense in the current context. This is particularly important in handling discourse, where the interpretation of a sentence containing such things as pronouns may depend in complex ways on the preceding discourse and knowledge of the subject matter.

This dual view of programs as data and data as programs would not have been possible in traditional programming languages. The special languages for expressing facts about grammar, semantics, and deduction are embedded in LISP, and share with it the capability of ignoring the artificial distinction between programs and data.

1.1.2 Organization of the Language Understanding Process

We can divide the process of language understanding into three main areas -- syntax, semantics, and inference. As mentioned above, these areas cannot be viewed separately but must be understood as part of an integrated system. Nevertheless, we have organized our programs along these basic lines, since each area has its own tools and concepts which make it useful to write special programs for it.

Listing these aspects of language understanding separately is somewhat misleading, as it is the interconnection and interplay between them which makes the system possible. Our parser does not parse a sentence, then hand it off to an interpreter. As it finds each piece of the syntactic structure, it checks its semantic interpretation, first to see if it is plausible, then (if possible) to see if it is in accord with the system's knowledge of the world, both specific and general. This has been done in a limited way by other systems, but in our program it is an integral part of understanding at every level.

A. Syntax

First we need a system for the syntactic analysis of input sentences, and any phrases and other non-sentences we might want in our dialogs. There have been many different parsing systems developed by different language projects, each based on a particular theory of grammar. The type of grammar chosen plays a major role in the type of semantic analysis which can be carried out. A language named PROGRAMMAR was designed specifically to fit the type of analysis used in this system. It differs from other parsers in that the grammar itself is written in the form of a collection of programs, and the parsing

system is in effect an interpreter for the language used in writing those programs.

Having chosen the "type of grammar", we need to formalize a grammar for parsing sentences in a particular language. Our system includes a comprehensive grammar of English following the lines of systemic grammar (see Section 2.3). This type of grammar is well suited to a complete language-understanding system since it views language as a system for conveying meaning and is highly oriented toward semantic analysis. It is intended to cover a wide range of syntactic constructions; one basic criterion for the completeness of the grammar is that a person with no knowledge of the system or its grammar should be able to type any reasonable sentence within the limitations of the vocabulary and expect it to be understood.

B. Inference

At the other end of the linguistic process we need a deductive system which can be used not only for such things as resolving ambiguities and answering questions, but also to allow the parser to use deduction in trying to parse a sentence. The system uses PLANNER, a deductive system designed by Carl Hewitt (see <Hewitt 1969, 1970>) which is based on a philosophy very similar to the general mood of this project. Deduction in PLANNER is not carried out in the traditional "logistic framework" in which a general procedure acts on a set of axioms or theorems expressed in a formal system of logic. Instead, each theorem is in the form of a program, and the deductive process can be directed to any desired extent by "intelligent theorems." PLANNER is actually a language for the writing of those theorems.

This deductive system must be given a model of the world, with the concepts and knowledge needed to make its deductions. Useful language-understanding can occur only when a program (or

person) has an adequate understanding of the subject he is talking about. We will not attempt to understand arbitrary sentences talking about unknown subjects, but instead will give the system detailed knowledge about a particular subject -- in this case, the simple robot world of children's toy blocks and some other common objects. The deductive system has a double task of solving goal-problems to plan and carry out actions for a robot within this world, and then talking about what it is doing and what the scene looks like. We want the robot to discuss its plans and actions as well as carry them out. We can ask questions not only about physical happenings, but also about the robot's goals. We can ask "Why did you clear off that block?" or "How did you do it?". This means that the model includes not only the properties of blocks, hands, and tables, but a model of the robot mind as well. We have written a collection of PLANNER theorems and data called BLOCKS, describing the world of toy blocks seen and manipulated by the robot, and the knowledge it needs to work with that world. (see Section 3.4). Figure 1 shows a typical scene.

C. Semantics

To connect the syntactic form of the sentence to its meaning, we need a semantic system which provides primitive operations relevant to semantic analysis. This includes a language in which we can easily express the meanings of words and syntactic constructions. The system includes mechanisms

The Robot's World

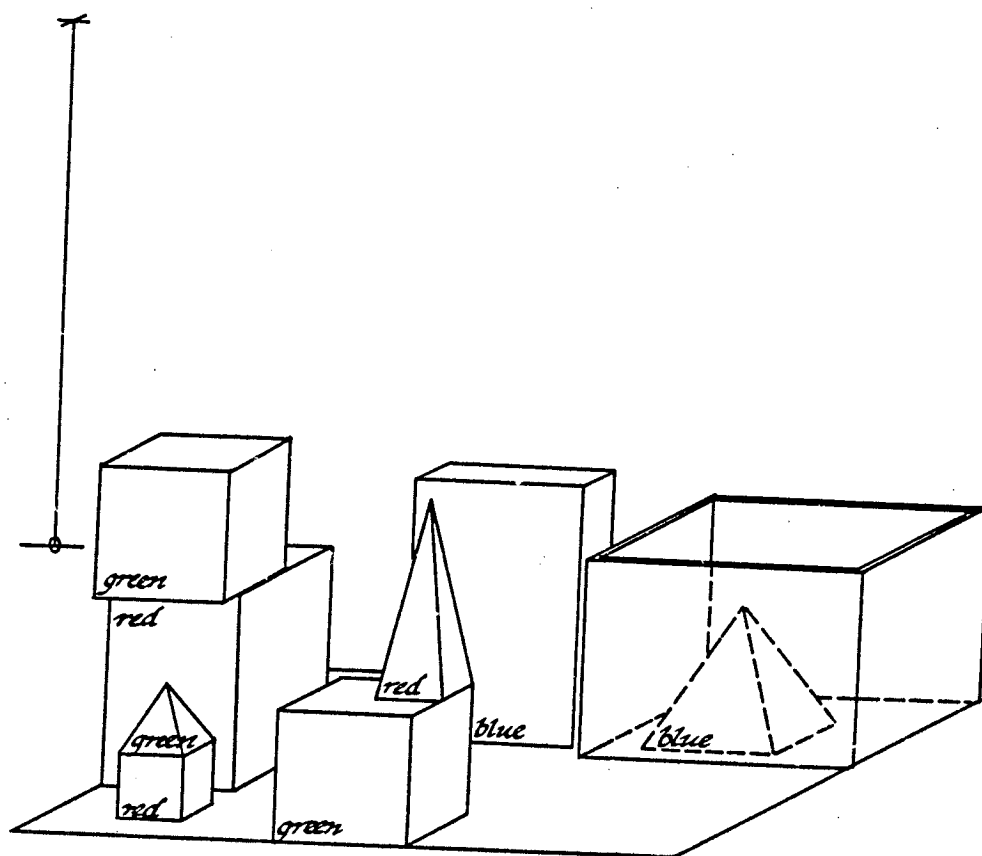


Figure 1

for setting up simple types of semantic networks and using deductions from them as a first phase of semantic analysis. For example, the network could include the information that a "block" is a physical object, while a "bloc" is a political object, and the definition of the word "support" could use this information in choosing the correct meanings for the sentences:

The red block supports the pyramid. and

The red bloc supports Egypt.

More important, the meaning of a word or construction is also defined in the form of a program to be interpreted in a semantic language. It is this procedural aspect of semantics which is missing in most other theories, which limit themselves to a particular type of network or relational structure. The meaning selected for a word can depend on any aspect of the sentence, the discourse, or the world. In deciding on the meaning of "one" in "Pick up the green one". we need a program which can examine past sentences. This program is included as part of the definition of the word "one".

The semantic system includes a powerful heuristic program for resolving ambiguities and determining the meaning of references in discourse. In almost every sentence, reference is made either explicitly (as with pronouns) or implicitly (as with the word "too") to objects and concepts not explicitly mentioned in that sentence. To interpret these, the program must have at its disposal not only a detailed grammatical analysis (to check for such things as parallel constructions), but also a powerful deductive capacity (to see which reference assignments are logically plausible), and a thorough knowledge of the subject it is discussing (to see which interpretations are reasonable in the current situation).

In order to deal with language in a human way, we must take into account all sorts of discourse knowledge. In addition to remembering the immediately previous sentences for such things as pronoun references, the system must remember what things have been mentioned throughout the discussion, so that a reference to "the pyramid" will mean "the pyramid we mentioned earlier" even if there are several pyramids in the scene.

In addition the system must have some knowledge of the way a person will communicate with it. If we ask "Is there a block on a green table?" "What color is it?", the word "it" refers to the block. But if we had asked "Is there a green block on a table?" "What color is it?", "it" must refer to the table since we would not ask a question which we had answered ourselves in the previous sentence.

Our semantic system works with a base of knowledge about simple semantic features in the subject domain, and with a collection of definitions for individual words. These definitions are written in a "semantics language" which allows simple words to be defined in a straightforward way, while allowing more complex words to call on arbitrary amounts of computation to integrate their meaning into the sentence.

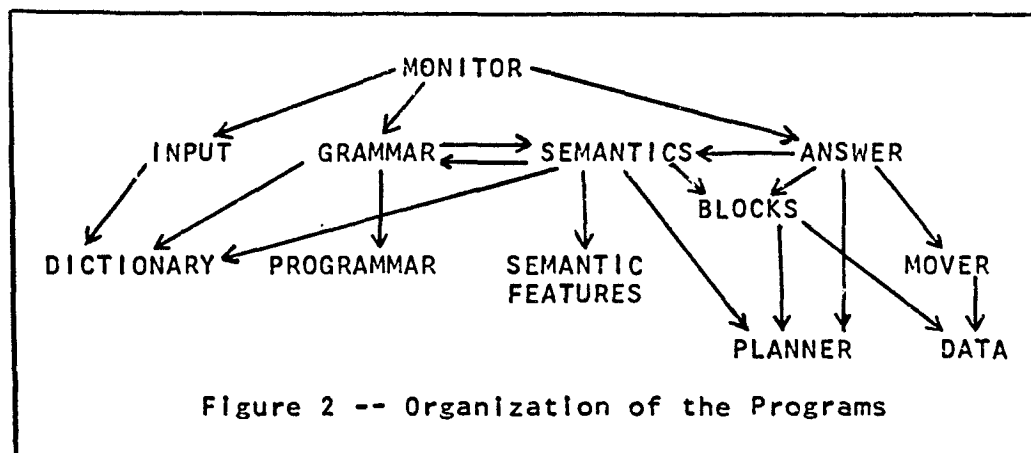
Finally we need a generative language capacity to produce answers to questions and to ask questions when necessary to resolve ambiguities. Grammatically this is much less demanding than the interpretive capacity, since humans can be expected to

understand a wide range of responses, and it is possible to express almost anything in a syntactically simple way. However, it takes a sophisticated semantic and deductive capability to phrase things in a way which is meaningful and natural in discourse, since the form of a response depends on both the context and on what the speaker assumes that the hearer knows and wants to know.

1.2 Implementation of the System

The language understanding program is written in LISP to run under the PDP-10 Incompatible Time-sharing System at the Artificial Intelligence Laboratory at MIT. When operating with a 200 word vocabulary and a fairly complex scene, it occupies approximately 80K of core. This includes the LISP interpreter, all of the programs, dictionary entries, and data, and enough free storage to remember a sequence of actions and to handle complex sentences and deductions. See Figure 3 for a more detailed description of memory usage.

The program is organized as indicated in Figure 2. (Arrows indicate that one part of the program calls another directly):



1. MONITOR is a small LISP program which calls the basic parts of the system. Since the system is organized vertically, most of the communication between components is done directly, and the monitor is called only at the beginning and end of the

	Parser	Semantics	Deduction	Other
Interpreters 26.1	PROGRAMMAR 5.8		PLANNER 5.5	LISP and Display 14.8
Knowledge of English 22.5	GRAMMAR 7.3	SEMANTICS 15.2		
Knowledge of Subject 16.5	DICTIONARY 1.7	DICTIONARY 6.0	BLOCKS 8.8	
Data for Scene 2.5			Assertions 1.3	Display 1.2
Total 67.6	14.8	21.2	15.6	16.0

Storage Allocation for Language Understanding Program
In Thousands of PDP-10 words

Note: Approximately 12 thousand additional words of free storage are necessary for a dialog like the one described in Section 1.3. As the length of dialog or complexity of the actions is increased, more free storage is needed.

Figure 2 -- Memory Requirements

understanding process.

2. INPUT is a LISP program which accepts typed input in normal English orthography and punctuation, looks up words in the dictionary, performs morphemic analysis (e.g. realizing that "running" is the "ing" form of the word "run", and modifying the dictionary definition accordingly), and returns a string of words, together with their definitions. This is the input with which the grammar works.

3. The GRAMMAR is the main coordinator of the language understanding process. It consists of a few large programs written in PROGRAMMAR to handle the basic units of the English language (such as clauses, noun groups, prepositional groups, etc.). There are two PROGRAMMAR compilers, one which compiles into LISP, which is run interpretively for easy debugging, and another which makes use of the LISP compiler to produce LAP assembly code for efficiency.

4. SEMANTICS is a collection of LISP programs which work in coordination with the GRAMMAR to interpret sentences. In general there are a few semantics programs corresponding to each basic unit in the grammar, each performing one phase of the analysis for that unit. These semantics programs call PLANNER to make use of deduction in interpreting sentences.

5. ANSWER is another collection of LISP programs which control the responses of the system, and take care of remembering the discourse for future reference. It contains a

number of heuristic programs for producing answers which take the discourse into account, both in deciding on an answer and in figuring out how to express it in fluent English.

6. PROGRAMMAR is a parsing system which interprets grammars written in the form of programs. It has mechanisms for building a parsing tree, and a number of special functions for exploring and manipulating this tree in the GRAMMAR programs. It is written in LISP.

7. The DICTIONARY actually consists of two parts. The first is a set of syntactic features associated with each word, used by the GRAMMAR. The second is a semantic definition for each word, written in a language which is interpreted by the SEMANTICS programs. The form of a word's definition depends on its word class (e.g. the definition of "two" is "2"). There are special facilities for irregular forms (like "geese" or "slept"), and only the definitions of root words are kept, since INPUT can analyze a variety of endings. The definitions are actually kept on the LISP property list of the word, and dictionary lookup is handled automatically by LISP.

8. The system has a network of SEMANTIC FEATURES, kept on property lists and used for an initial phase of semantic analysis. The features subdivide the world of objects and actions into simple categories, and the semantic interpreter uses these categories to make some of its choices between alternative definitions for a word (for example "in" would mean

"contained in" when applied to a hollow object like a box, but would mean "part of" when applied to a conglomerate like a stack, as in "the red block in the tall stack").

9. BLOCKS is a collection of PLANNER theorems which contain the system's knowledge about the properties of the physical world, and the methods of achieving goals and making deductions about it. It is a very simple model, designed less for a realistic simulation of a robot, and more to give the system a world to talk about in which one can say many different types of things.

10. MOVER is a set of display routines written in LAP which simulate the robot's world on the DEC 340 graphic display attached to the PDP-10 computer. It displays the scene indicated by the DATA currently in memory, and any action called for is displayed in real time, so the human can get visual feedback from his commands to the robot. The input sentences and responses are also displayed. Figure 1 displays a typical scene.

11. PLANNER is the deductive system used by the program, and is used at all stages of the analysis, both to direct the parsing process and to deduce facts about the BLOCKS world. Currently the system uses Micro-Planner, an abbreviated version of Carl Hewitt's original language, implemented by Terry Winograd, Gerald Sussman, and Eugene Charniak (Sussman). It is written in LISP.

12. The DATA about the current scene is in the form of PLANNER assertions describing the objects, their sizes, shapes, colors, and locations. The system assumes it has some sort of robot "eye", so all of this information is initially known and does not have to be entered in English.

When running compiled, the system is fast enough to carry on a real-time discourse. Each sentence takes from 5 to 20 seconds to analyze and respond to, and the display is designed to move at the speed of a real arm.

The program listings are all available in hard copy and DECTape, but before implementing them on another system, it would be wise to read about the special features of MACLISP <White> which may be in conflict with other LISP implementations.

1.3 Sample Dialog

This section contains a sample of a dialog with the language understanding system. It carries out actions and answers questions about a simple scene containing a table, hand, and box, and several blocks and pyramids. The lines in lower case were typed in by a person, and the lines in upper case are the computer's response. Indented lines are comments describing some of the features being exhibited.

The dialog was carried out in real time with the response of the "robot" displayed on a CRT screen. Figures are included to show the state of the scene at various points.

1. pick up a big red block.

OK.

The system answers "OK" when it carries out a command. In order to pick up the red block, it had to clear it off by finding a space for the green one and moving the green one away. Figure 4 shows the scene when the command was given, Figure 5 shows the green block being put out of the way, and Figure 6 shows the end result.

2. grasp the pyramid.

I DON'T UNDERSTAND WHICH PYRAMID YOU MEAN.

There are three pyramids in the scene, so the program does nothing.

3. find a block which is taller than the one you are holding and put it into the box.

The system has to figure out what is meant by "one" and "it", as well as using deduction to decide which block it is holding.

Pick up a big red block

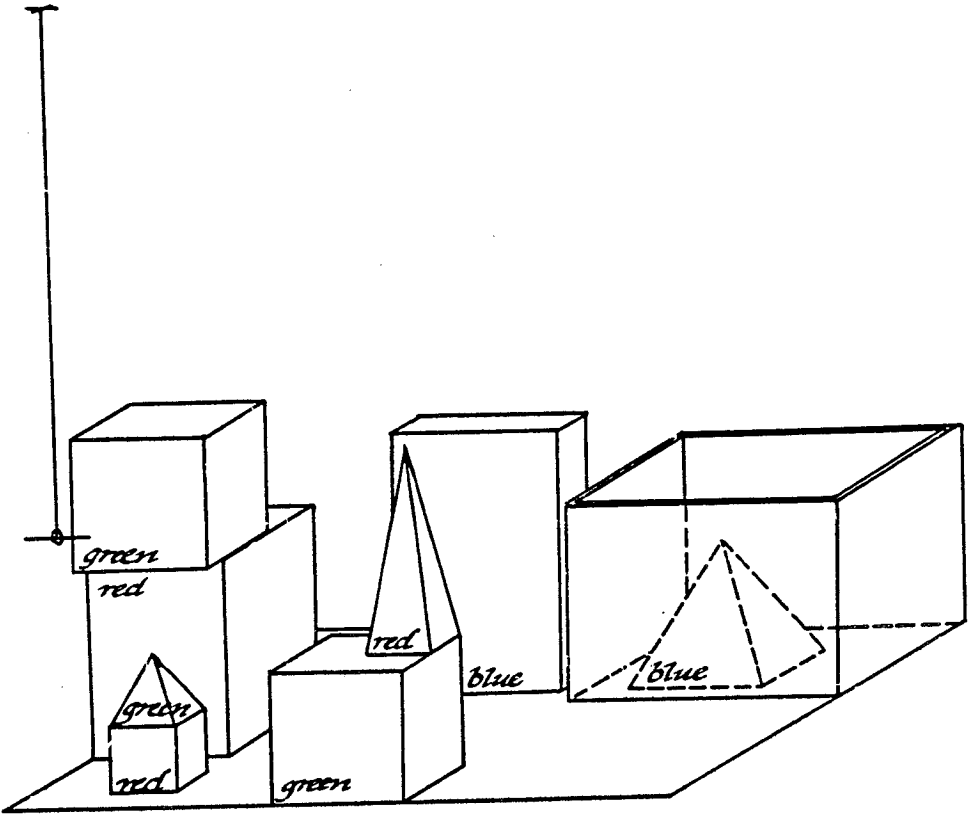


Figure 4

Pick up a big red block.

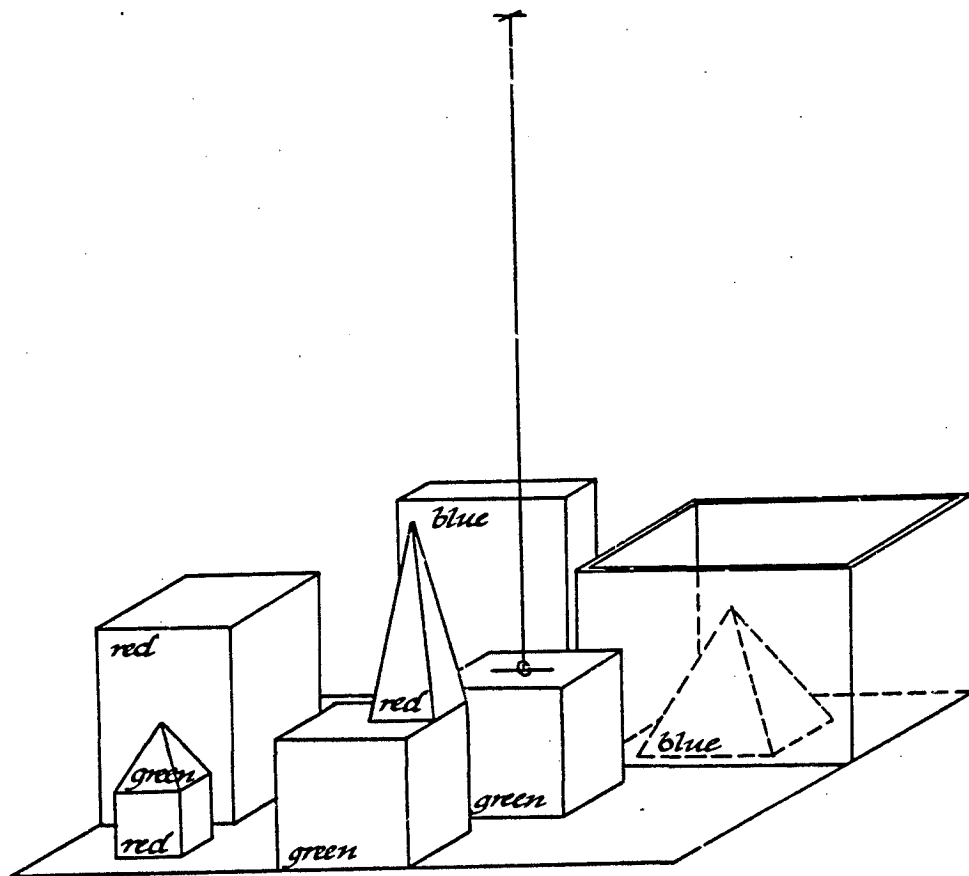


Figure 5

Pick up a big red block.

OK.

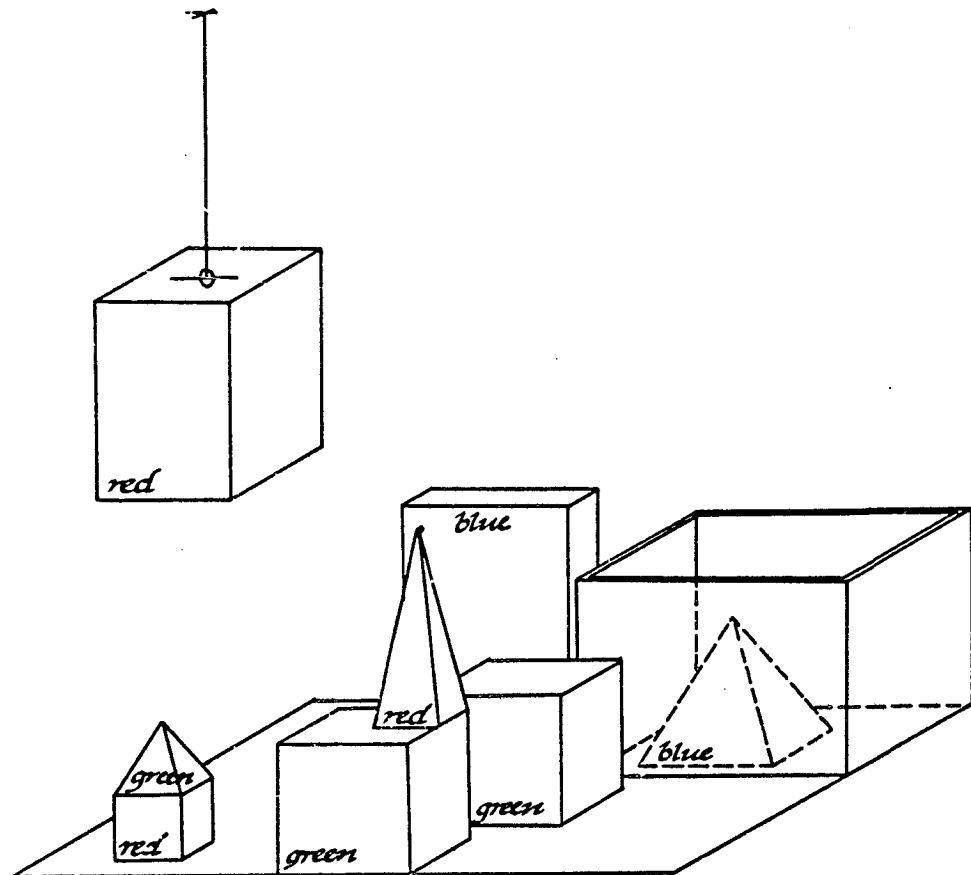


Figure 6

BY "IT", I ASSUME YOU MEAN THE BLOCK WHICH IS TALLER THAN THE ONE I AM HOLDING.

OK.

Printed out to keep us informed.

The hand has to set down the red block to pick up the taller blue one. Figure 7 shows the result.

4. what does the box contain?

THE BLUE PYRAMID AND THE BLUE BLOCK.

5. what is the pyramid supported by?

Earlier the system responded to "the pyramid" by saying "I don't understand which pyramid you mean." now, it knows that we must mean the one which was just mentioned.

THE BOX.

6. how many blocks are not in the box?

Negatives such as "not" and "no" are handled whether they appear in clauses or noun groups (such as "no boxes").

FOUR OF THEM.

7. Is at least one of them narrower than the one which I told you to pick up?

The phrase "at least one of them" must be analyzed to refer back to the previous question, and "the one which..." must also refer to blocks. Noun groups can have complicated numbers such as "at least one", "more than three", "exactly seven", etc. The memory of past events is used to deduce which block is "the one which I told you to pick up". Notice the complicated grammatical structure in which "the one" is actually the object of a clause embedded two levels deeper.

YES, THE RED CUBE.

8. Is it supported?

*Find a block which is taller than the one
you are holding and put it into the box.*

*By "it", I assume you mean the block
which is taller than the one I am holding.
OK.*

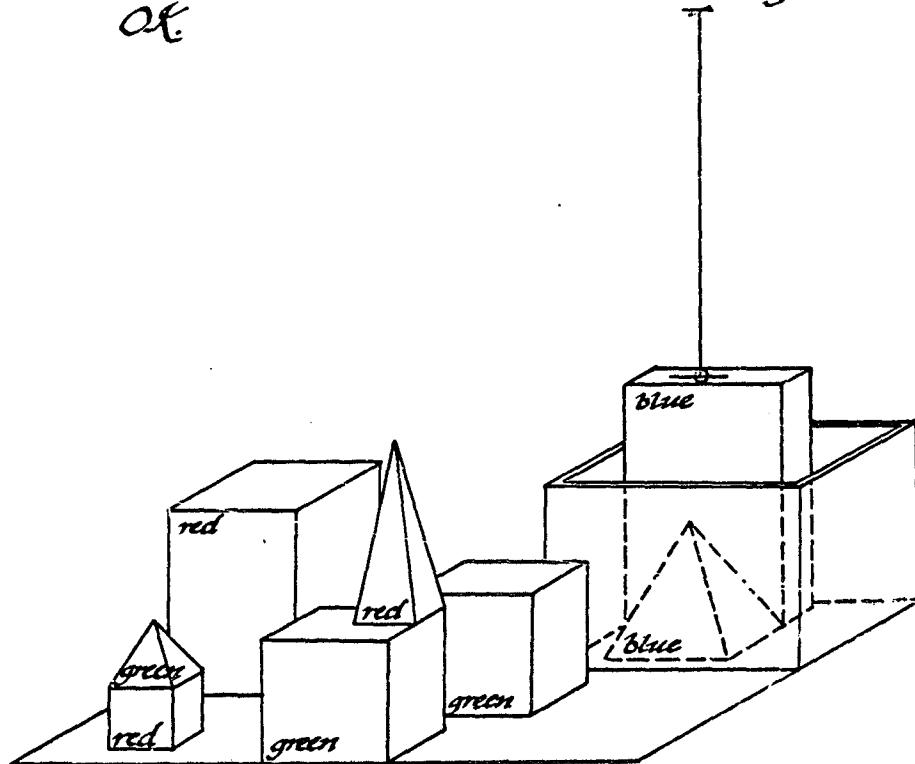


Figure 7

Passives can be used without a "by ..." phrase.

YES, BY THE TABLE.

9. can the table pick up blocks?

NO.

The semantic rules make this a nonsensical question which therefore must be false.

10. can a pyramid be supported by a block?

YES.

The deductive system finds an actual example, so it knows this is possible.

11. can a pyramid support a pyramid?

I DON'T KNOW.

The question makes sense, but there is no example in the scene, so there is no easy way to tell without being able to examine the robot's programs.

12. stack up two pyramids.

I CAN'T.

The robot tries and fails.

13. the blue pyramid is nice.

I UNDERSTAND.

This data is added to the data base.

14. I like blocks which are not red, but I don't like anything which supports a pyramid.

I UNDERSTAND.

Two new theorems are created for proving things about "liking".

15. do I like the box?

NO.

The box supports whatever it contains, including a pyramid.

16. do I like anything in the box?

YES, TWO THINGS: THE BLUE BLOCK AND THE BLUE PYRAMID.

Presumably we like anything which is nice, and the blue block is certainly not red.

17. will you please stack up both of the red blocks and either a green cube or a pyramid?

OK.

Logical connectives such as "and" "or" "either" etc. are handled in both the grammar and semantics. A command is executed even if it looks like a question. The robot carries out the command by the series of steps shown in figures 8 through 11. It puts a green cube on the large red block (note that it chooses the green cube with nothing on it), then removes the small pyramid from the little red cube, so that it can use that cube to complete the stack as specified.

18. which cube is sitting on the table?

THE LARGE GREEN ONE WHICH SUPPORTS THE RED PYRAMID.

Ordinarily the system would be satisfied with calling this "a big green cube", but since we asked "which cube" it has to be more specific in naming it. The use of substitute nouns (such as "one") and pronouns in answers makes the discourse more natural.

19. Is there a large block behind a pyramid?

YES, THREE OF THEM: A LARGE RED ONE, A LARGE GREEN CUBE AND THE BLUE ONE.

20. put a small one onto the green cube which supports a pyramid.

"A small one" must contrast with the earlier phrase "a large block", so this means "a small block".

OK.

21. put the littlest pyramid on top of it.

*Will you please stack up both of the red
blocks and either a green cube or a
pyramid.*

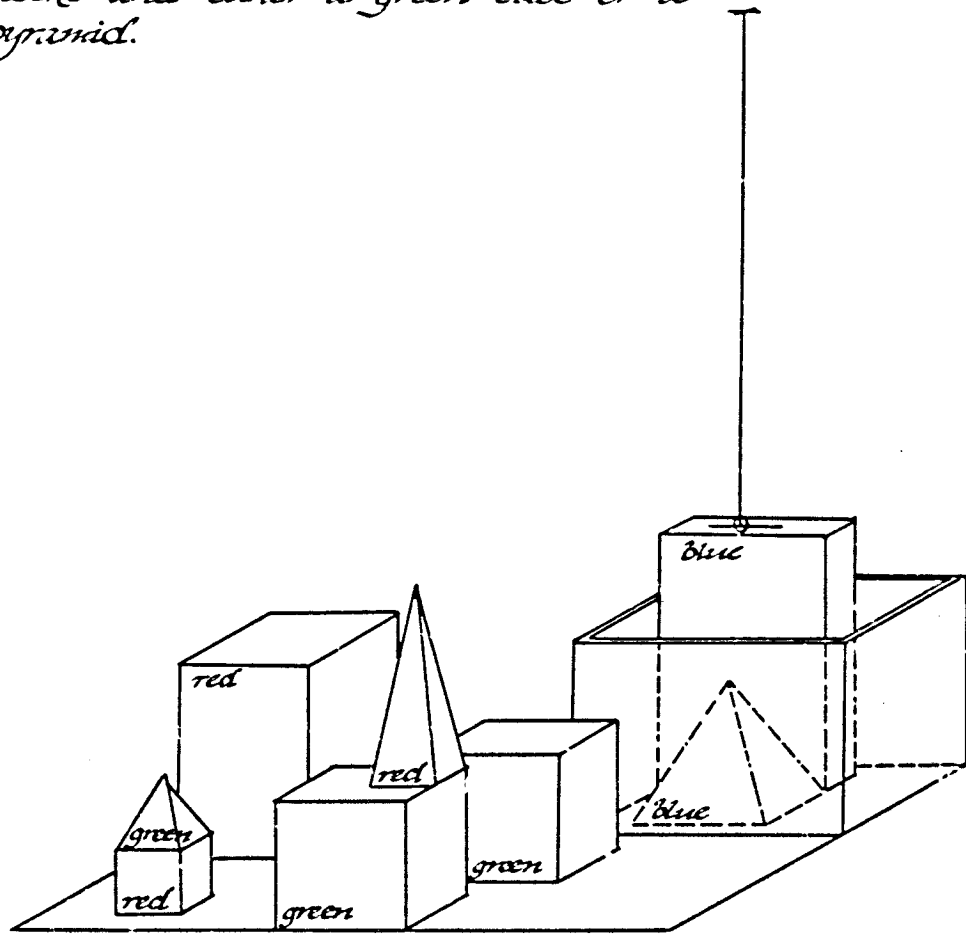


Figure 8

*Will you please stack up both of the red
blocks and either a green cube or a pyramid.*

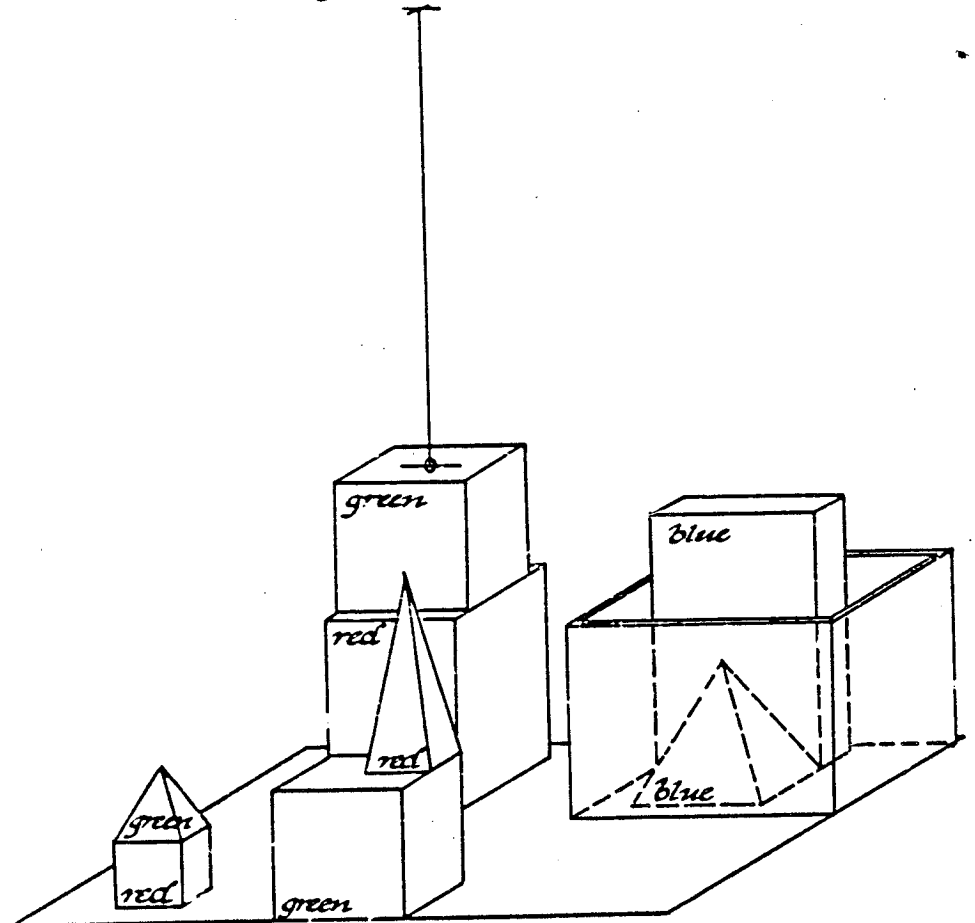


Figure 9

Will you please stack up both of the red blocks and either a green cube or a pyramid.

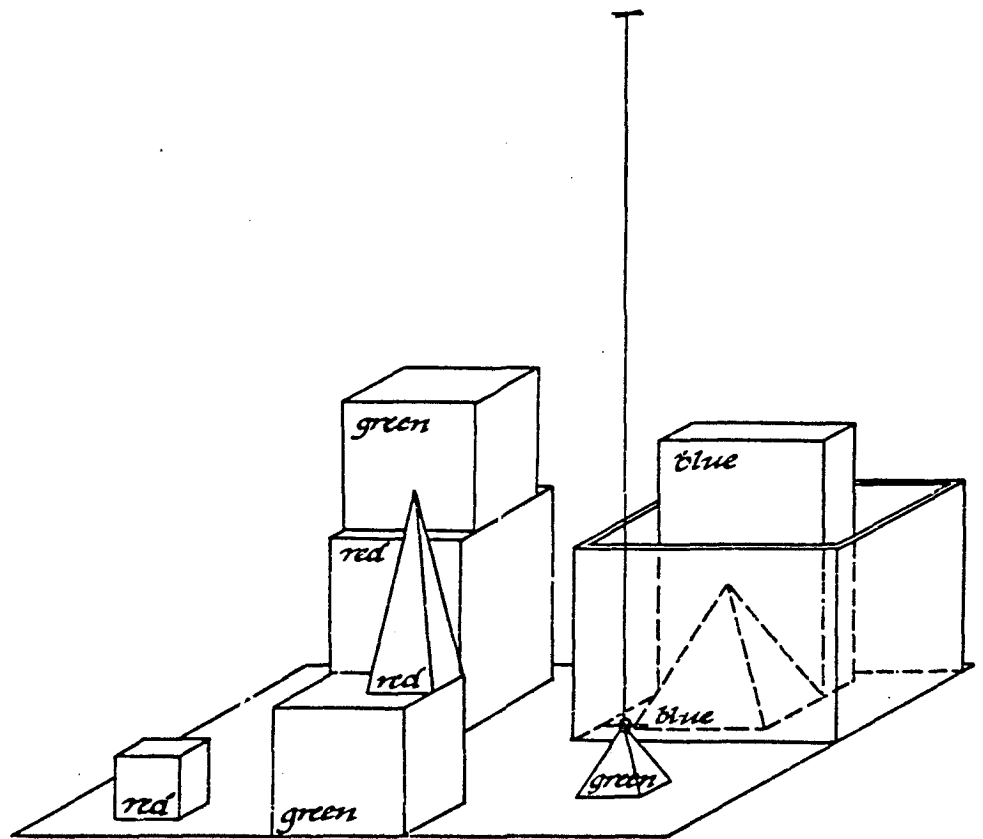


Figure 10

*Will you please stack up both of the red
blocks and either the green cube or a pyramid.*

OK.

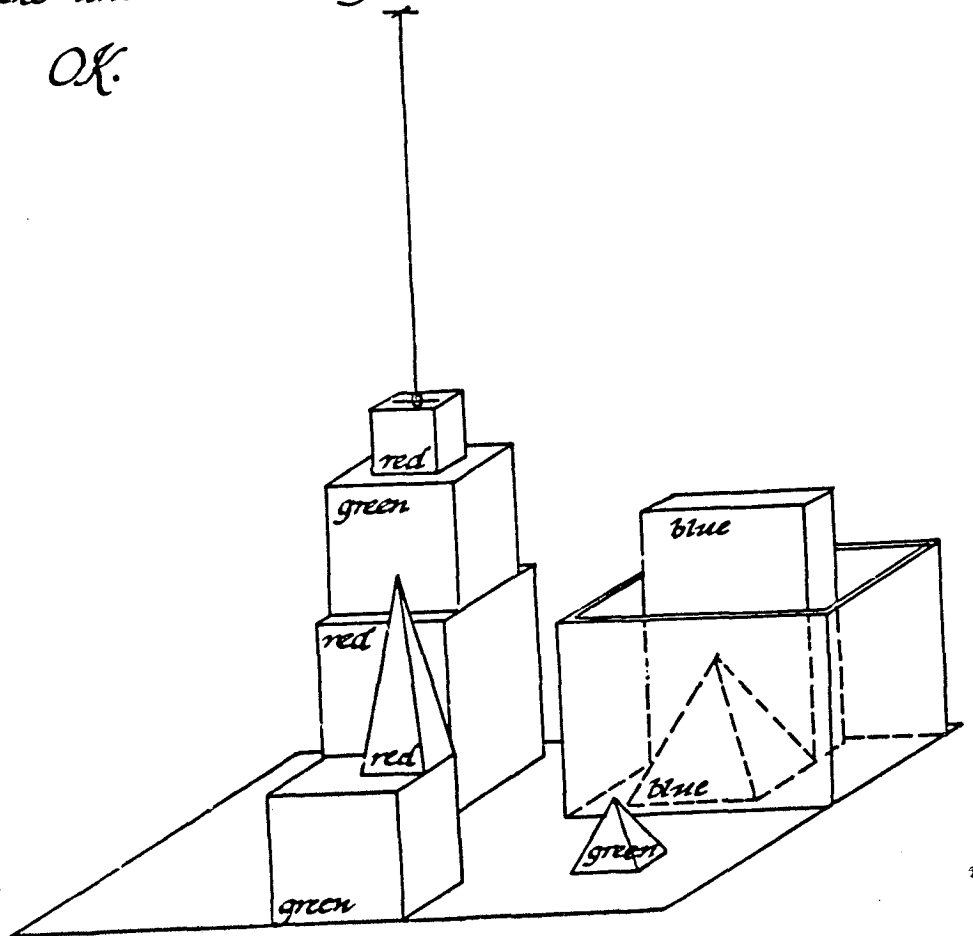


Figure 11

Put the littlest pyramid on top of it.

OK.

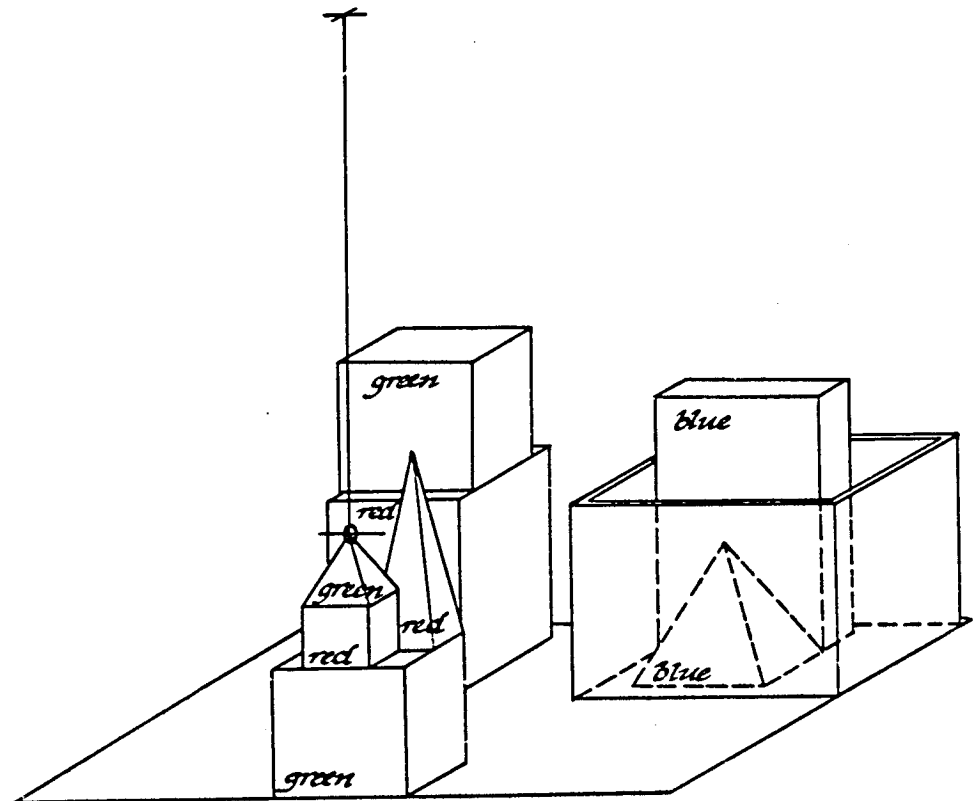


Figure 12

Words like "littlest" are not in the dictionary but are interpreted from the root forms like "little". The scene now looks like Figure 12.

OK.

22. does the shortest thing the tallest pyramid's support supports support anything green?

The grammar can handle words like "support" which are both verbs and nouns without getting confused, and can understand possessives. To compute the answer, see Figure 13 for the scene.

YES, THE GREEN PYRAMID.

23. what color is it?

BY "IT", I ASSUME YOU MEAN THE SHORTEST THING THE TALLEST PYRAMID'S SUPPORT SUPPORTS.

If we had asked "what size is it?", the system would have assumed that "it" meant the green pyramid, but nobody would ask the color of something which was just called "the green pyramid", so it had to look farther for the reference of "it".

RED

24. how many things are on top of green cubes?

I'M NOT SURE WHAT YOU MEAN BY "ON TOP OF" IN THE PHRASE "ON TOP OF GREEN CUBES".

DO YOU MEAN:

- 1 - DIRECTLY ON THE SURFACE
- 2 - ANYWHERE ON TOP OF ?

When there are several interpretations and none is clearly better, the system has to ask the user for more information. The paraphrases were included with the dictionary definition of "on", and the request for more information occurs automatically when the heuristics for resolving ambiguities aren't sufficient. A request like this can also occur when the meaning of a pronoun such as "it" or "they" is unclear.

Does the shortest thing the tallest pyramid's support supports anything green?

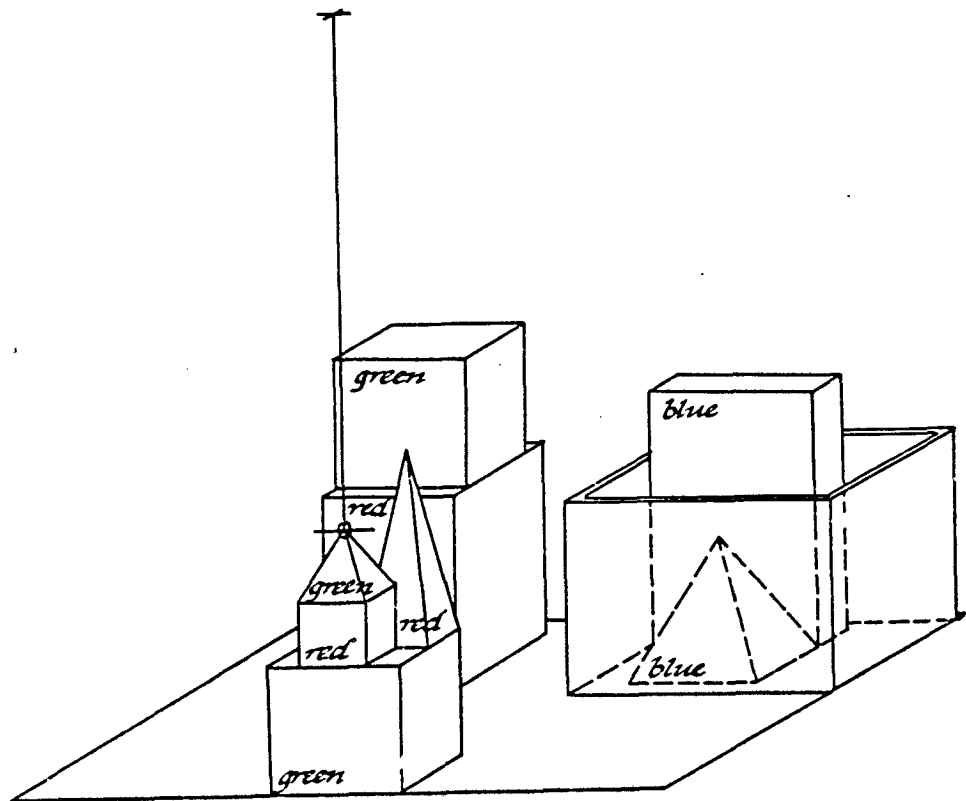


Figure 13

2

THREE OF THEM.

25. had you touched any pyramid before you put the green one on the little cube?

YES, THE GREEN ONE.

The system deduces from its memory what event is being referred to, and can relate other events to it in time. The analysis includes complex tenses of verb groups, and there are special facilities for keeping track of the various forms of irregular verbs like "have".

26. when did you pick it up?

WHILE I WAS STACKING UP THE RED CUBE, A LARGE RED BLOCK AND A LARGE GREEN CUBE.

A time is described by naming the main goal which was being worked on.

27. why?

The system remembers immediately previous sentences to understand questions like this and the following ones.

TO GET RID OF IT.

By keeping track of selected parts of the original subgoal tree, the system has some understanding of its own motives.

28. why did you do that?

TO CLEAN OFF THE RED CUBE.

"That" refers to the action mentioned in the answer.

29. why did you clear off that cube?

Here, "that" means the one just referred to.

TO PUT IT ON A LARGE GREEN CUBE.

30. why did you do that?

TO STACK UP THE RED CUBE , A LARGE RED BLOCK AND A LARGE GREEN CUBE.

31. why did you do that?

BECAUSE YOU ASKED ME TO.

It remembers what it did, not how the request was worded.

32. how did you do it?

"It" refers to the action mentioned in the last question, rather than the answer.

BY PUTTING A LARGE RED BLOCK ON THE TABLE ; THEN LETTING GO OF IT ; THEN PUTTING A LARGE GREEN CUBE ON IT ; THEN LETTING GO OF THAT CUBE ; THEN PUTTING THE RED CUBE ON THAT CUBE ; THEN LETTING GO OF THAT CUBE.

This is a description of the top-level subgoals used to achieve the goal.

33. how many objects did you touch while you were doing it?

Time references can refer to a time span, using words like "while".

FOUR OF THEM.

34. what did the red cube support before you started to clean it off?

THE GREEN PYRAMID.

By remembering what has been moved, it is possible to deduce things about earlier states of the scene. We can use complex nestings of clauses like "before you started to clean it off".

35. there were five blocks to the left of the box then.

"Then" must refer to the time mentioned in the last sentence.

NO, ONLY FOUR OF THEM: THE RED CUBE , TWO LARGE GREEN CUBES AND A LARGE RED BLOCK.

Earlier in the dialog, information about "liking" was accepted at face value. But the system knows that it has complete information about about the basic properties of objects, including location. It checks any statements about location or other such properties to see whether it agrees or disagrees.

36. put the blue pyramid on the block in the box.

This is syntactically ambiguous, but is understood without ambiguity since the parsing corresponding to "put the blue pyramid which is on the block in the box." does not make sense in the current scene, while "put the blue pyramid on the block which is in the box" does. We can see the situation in Figure 14, and the result in Figure 15.

OK.

37. Is there anything which is bigger than every pyramid but is not as wide as the thing that supports it?

The system understands quantifiers such as "every", "any", "some", etc., and structures such as "bigger than", "as wide as", etc. Here "it" refers to a noun group within the same sentence but two levels up in the parsing tree.

YES, THE BLUE BLOCK.

39. does a steeple

SORRY, I DON'T KNOW THE WORD "STEEPLE". PLEASE TYPE TWO SPACES.

The system interrupts immediately when it sees a word it can't figure out. The two spaces are to clear the input buffer of anything typed after the message was sent but before the typist saw it.

40. a "steeple" is a stack which contains two green cubes and a pyramid.

The quote marks indicate that we are defining a new word.

Put the blue pyramid on the block in the box.

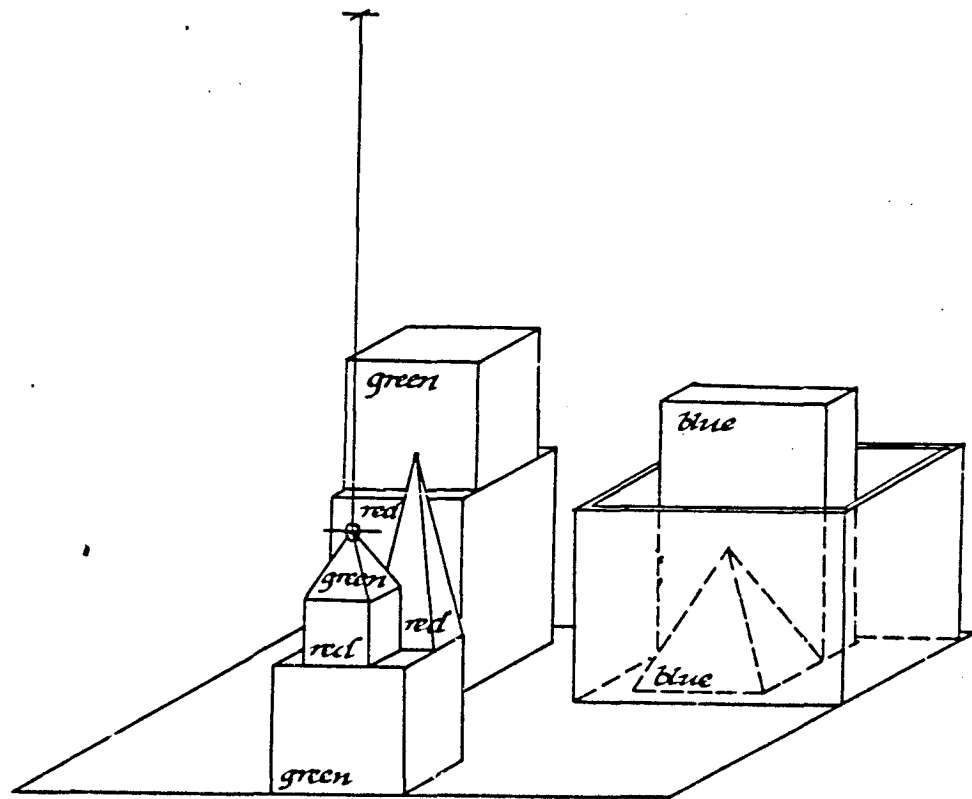


Figure 14

Put the blue pyramid on the block in the box.

OK.

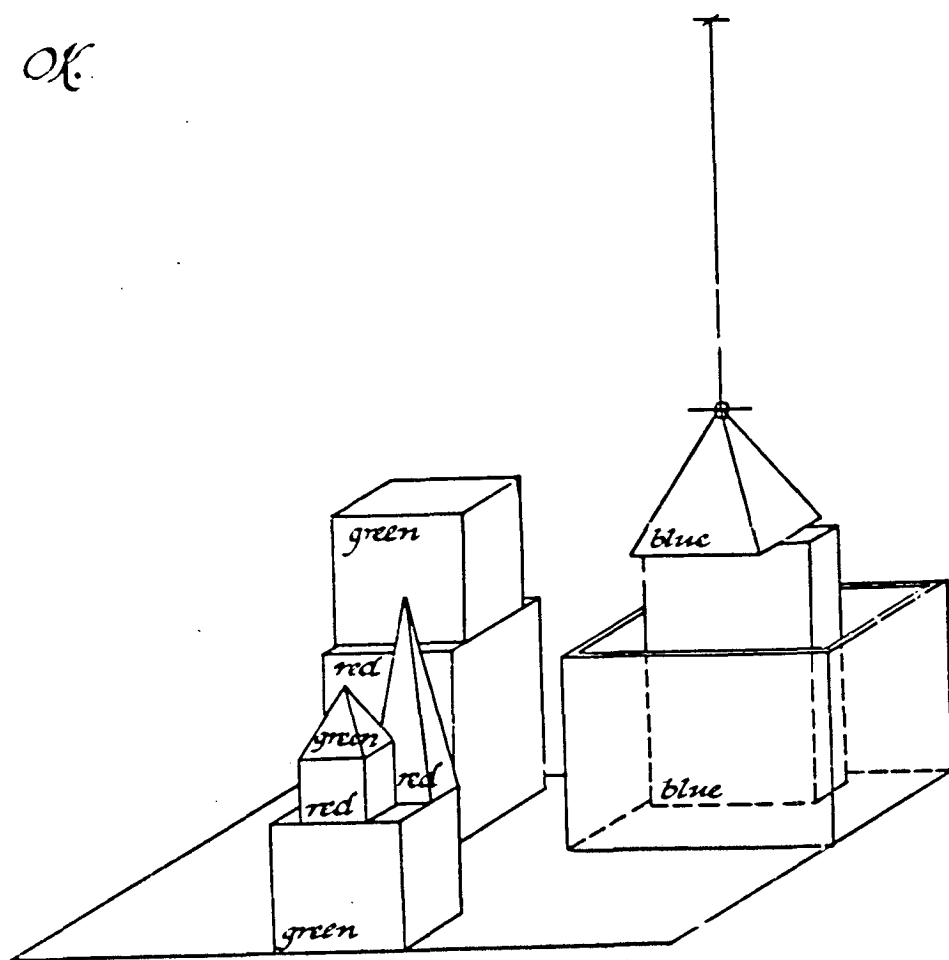


Figure 15

I UNDERSTAND.

41. are there any steeples now?

NO.

The new definition is used as a description to recognize steeples.

42. build one.

OK.

The description given in the definition is enough to deduce a plan for building new objects as well as recognizing them. The word "one" means "a steeple", and the plan is carried out. First, one of the large green cubes is cleared off by removing the red cube and two pyramids. Then the two green cubes and the tall red pyramid are stacked up to fit the description. Figures 16 through 19 show the initial state, some intermediate steps, and the result.

43. call the biggest block "superblock".

OK.

We can also give objects names.

44. have you picked up superblock since we began?

YES.

45. why did you drop it?

BY "IT", I ASSUME YOU MEAN THE SUPERBLOCK.

I CAN'T EXPLAIN A NON-EXISTENT EVENT.

There is no record in memory of any block being dropped.

46. is there anything to the right of the red pyramid?

YES, FOUR THINGS: THE BOX, THE RED CUBE, THE BLUE PYRAMID AND THE BLOCK NAMED SUPERBLOCK

Once an object has a name, the name is used to help identify it in responses.

Build one.

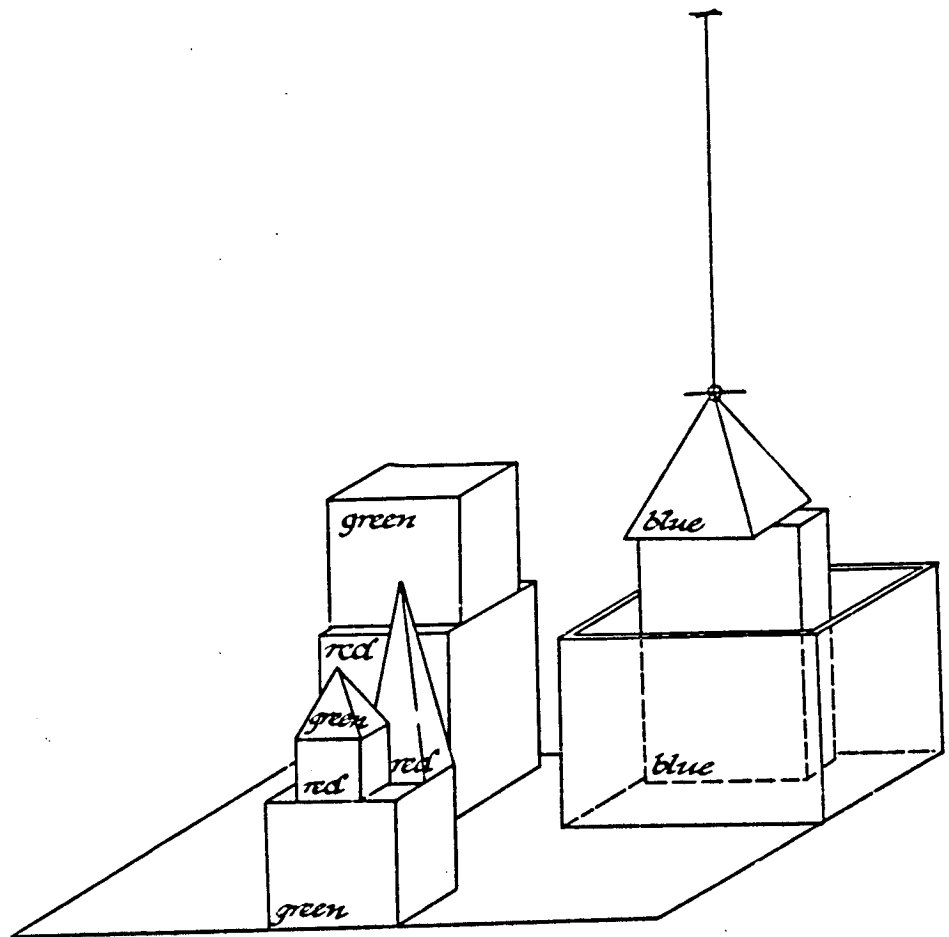


Figure 16

Build one.

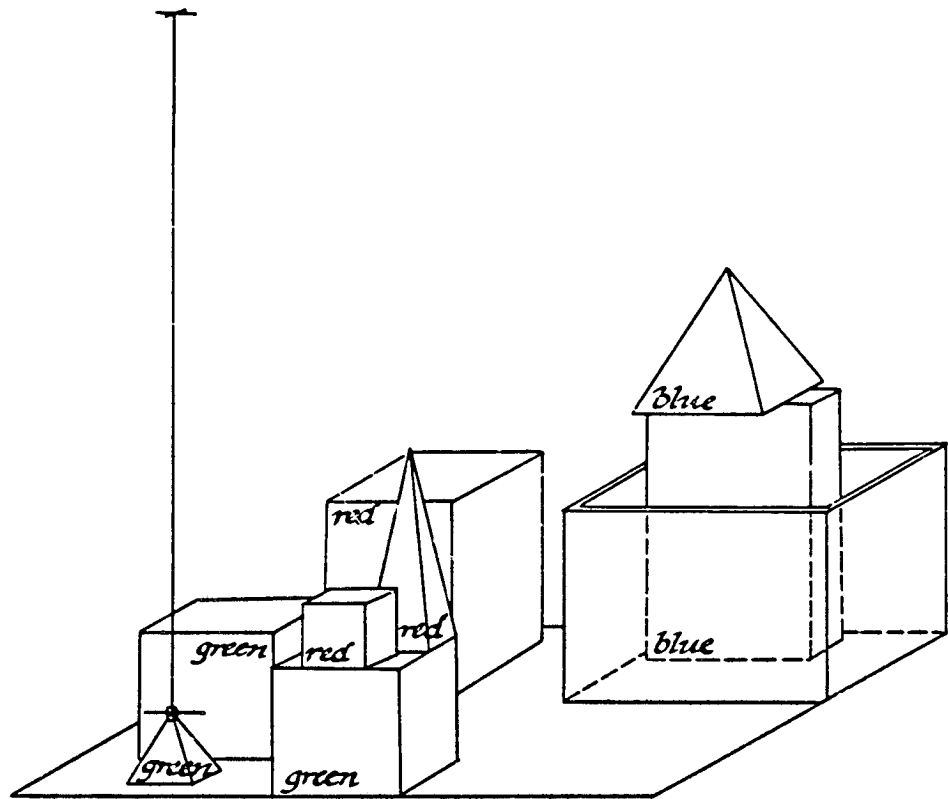


Figure 17

Build one.

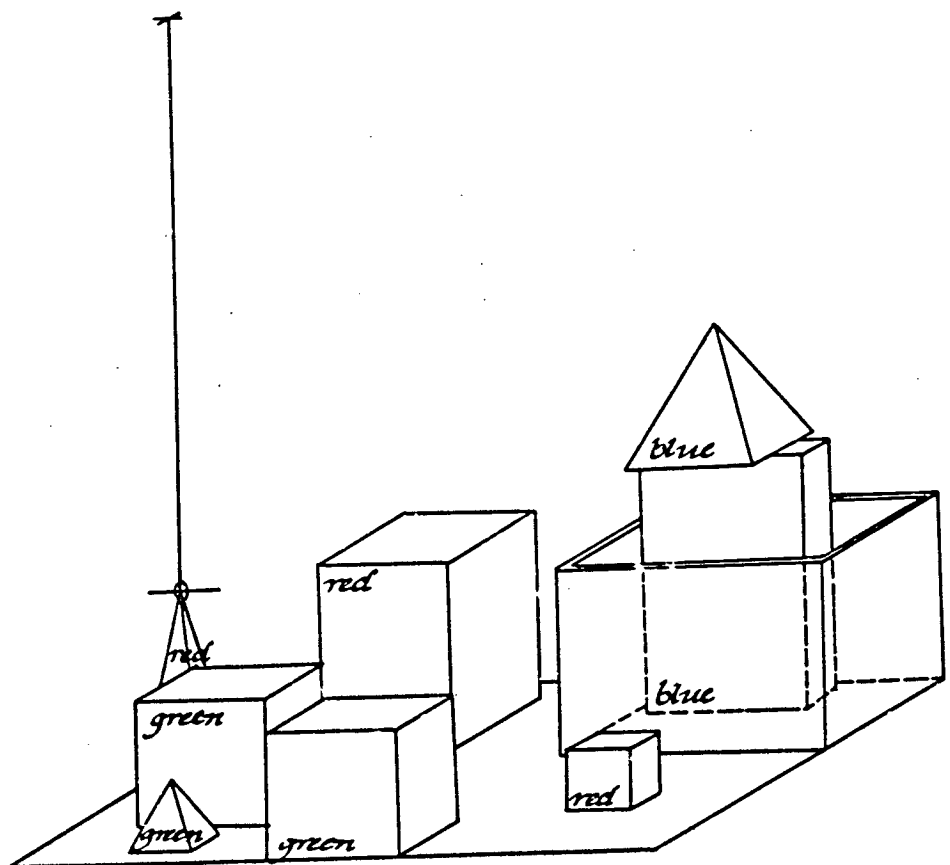


Figure 18

Build one.

OK

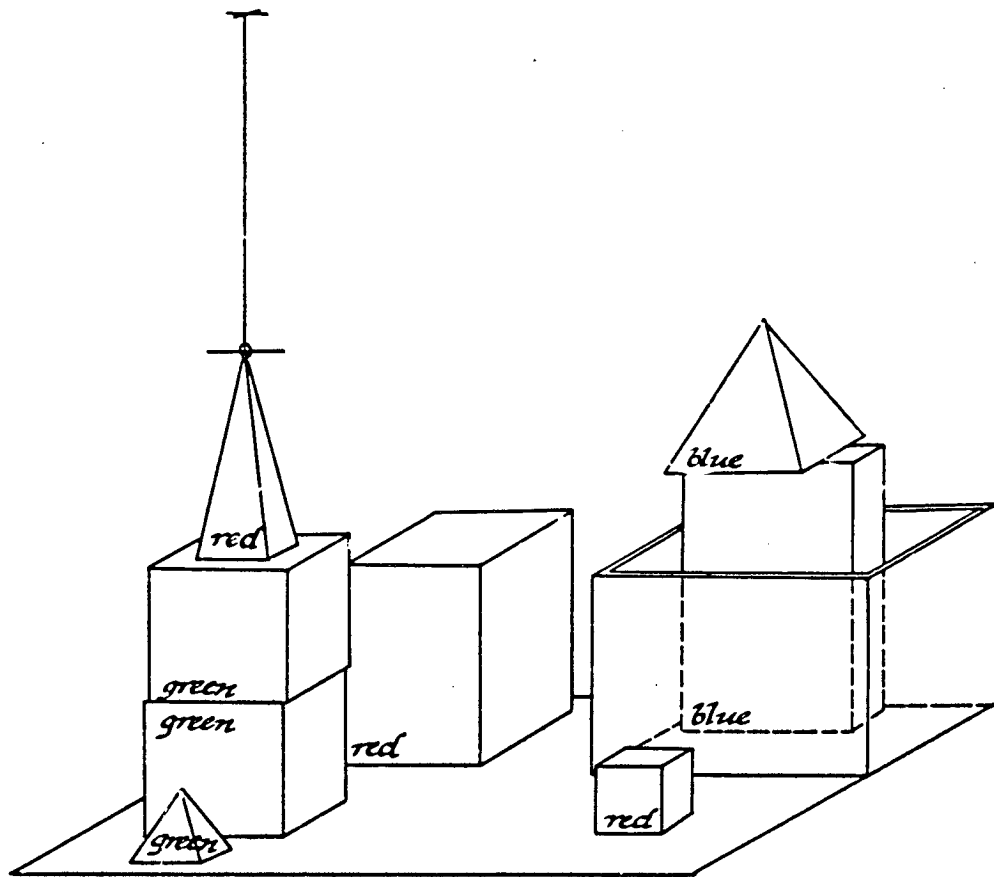


Figure 19

47. thank you.

YOU'RE WELCOME!

The special idiom-defining capabilities allow us to include such pleasantries without modifying the grammar.

Chapter 2. Syntax

2.1 Basic Approach to Syntax

2.1.1 Syntax and Meaning

The decision to consider syntax as a proper study devoid of semantics is a basic tenet of most current linguistic theories. Language is viewed as a way of organizing strings of abstract symbols, and competence is explained in terms of symbol-manipulating rules. At one level this has been remarkably successful. Rules have been formulated which describe in great detail how most sentences are put together. The problem comes in trying to relate those symbols and structures to the function of language as a vehicle for conveying meaning. The same approach which has worked so well in accounting for the machinations of syntax has been unable to provide any but the most rudimentary and unsatisfactory accounts of semantics.

The problem is not that current theories are finding wrong answers to the questions they ask; it is that they are asking the wrong questions. What is needed is an approach which can deal meaningfully with the question "How is language organized to convey meaning?" rather than "How are syntactic structures organized when viewed in isolation?".

How does a sentence convey meaning beyond the meanings of individual words? Here is the place for syntax. The structure of a sentence can be viewed as the result of a series of grammatical choices made in generating it. The speaker encodes meaning by choosing to build the sentence with certain "features", and the problem of the hearer is to recognize the

presence of those features and interpret their meaning.

We want to analyze the possible choices of features and functions which grammatical structures can have. For example, we might note that all sentences must be either IMPERATIVE, DECLARATIVE, or a QUESTION, and that in the last case they must choose as well between being a YES-NO question or a WH- question containing a word such as "why" or "which". We can study the way in which these features of sentences are organized -- which ones form mutually exclusive sets (called "systems"), and which sets depend on the presence of other features (like the set containing YES-NO and WH- depends on the presence of QUESTION). This can be done not only for full sentences, but for smaller syntactic units such as noun groups and prepositional groups, or even for individual words.

In addition we can study the different functions a syntactic "unit" can have as a part of a larger unit. In "Nobody wants to be alone.", the clause "to be alone" has the function of OBJECT in the sentence, while the noun group "nobody" is the SUBJECT. We can note that a transitive clause must have units to fill the functions of SUBJECT and OBJECT, or that a WH- question has to have some constituent which has the role of "question element" (like "why" in "Why did he go?" or "which dog" in "Which dog stole the show?").

In most current theories, these features and functions are implicit in the syntactic rules. There is no explicit mention

of them, but the rules are designed in such a way that every sentence will in fact be one of the three types listed above, and every WH- question will in fact have a question element. The difficulty is that there is no attempt in the grammar to distinguish significant features such as these from the infinite number of other features we could note about a sentence, and which are also implied by the rules.

If we look at the "deep structure" of a sentence, again the features and functions are implicit. The fact that it is a YES-NO question is indicated by a question marker hanging from a particular place in the tree, and the fact that a component is the object or subject is determined from its exact relation to the branches around it. The problem isn't that there is no way to find these features in a parsing, but that most theories don't bother to ask "Which features of a syntactic structure are important to conveying meaning, and which are just a by-product of the symbol manipulations needed to produce the right word order."

What we would like is a theory in which these choices of features are primary. Professor M.A.K. Halliday at the University of London has been working on such a theory, called Systemic Grammar (see references <Halliday 1961, 1966a, 1966b, 1967> <Huddleston>, <Hudson>). His theory recognizes that meaning is of prime importance to the way language is structured. Instead of having a "deep structure" which looks like a kind of syntactic structure tree, he deals with "system networks" which describe the way different features interact and depend on each other. The primary emphasis is on analyzing the limited and highly structured sets of choices which are made in producing a sentence or constituent. The exact way in which these choices are "realized" in the final form is a necessary but secondary part of the theory.

The realization rules carry out the work which would be done by transformations in transformational grammar (TG). In TG, the sentences "Sally saw the squirrel.", "The squirrel was seen by Sally.", and "Did Sally see the squirrel?" would be derived from almost identical deep structures, and the difference in final form is produced by transformations. In systemic grammar, these would be analyzed as having most of their features in common, but differing in one particular choice, such as PASSIVE vs. ACTIVE, or DECLARATIVE vs. QUESTION. The realization rules would then describe the exact word order used to signal these features.

What does this theory give us to use in a language understanding program? What kinds of parsings does it produce? If we look at a typical parsing by a systemic grammar, we note several points. First, it is very close to the surface structure of the sentence. There is no rearrangement into supposed "underlying" forms. Instead, each constituent is marked with features indicating its structure and function. Instead of saying that "Did John go?" has an underlying structure which looks like "John went.", we simply note that it has the features QUESTION and YES-NO, and that the noun group "John" has the function SUBJECT. Other parts of the language understanding process do not have to be concerned with the exact way the parsing tree is structured, since they can deal directly with the relevant features and functions.

What is more important is that these features are not random unrelated lists of observations. They are part of a highly structured network, and the grammatical theory includes a description of that network. When we do semantic analysis, we are not faced with the task of inventing "projection rules" to deal with the raw form of specific syntactic rules. Instead we can ask "what aspect of meaning does this system convey?", and "What is the significance of this particular feature within its system?".

2.1.2 Parsing

In implementing a systemic grammar for a computer program for understanding language, we are concerned with the process of recognition rather than that of generation. We do not begin with choices of features and try to produce a sentence. Instead we are faced with a string of letters, and the job is to recognize the patterns and features in it. We need the inverse of realization rules -- interpretation rules which look at a pattern, identify its structure, and recognize its relevant features. This interpretation process is closely related to other types of pattern recognition, and many interesting parallels can be drawn with the process of interpreting a visual scene (see <Winograd 1969>). The important aspect of both types of interpretation is looking for symbolic features which will be relevant to understanding, so that the parsing can be integrated with the rest of the understanding process. In general, this problem of isolating important features from complex information and representing them symbolically is a central issue for Artificial Intelligence, and the idea of a "systemic" parser may be of use in other areas.

The parsing system for our program is actually an interpreter for PROGRAMMAR, a language for writing grammars. It is basically a top-down left-to-right parser, but it modifies these properties when it is advantageous to do so. By writing in a language designed for grammars, we can express the

regularities of language in a straightforward way, as simply as in a syntax-directed parser. The primitives of the language are those operations which have particular relevance to natural language and its parsing.

A program for parsing language is as much a "generative" description of the language as is a set of rules for producing sentences. The meaning of "generative" in Chomsky's original sense (<Chomsky 1957, 1965>) is that the grammar should associate a structural description to each permissible sentence in the language. A parsing program does just that.

By making the formalism for grammars a programming language, we enable the grammar to use special tools to handle complex constructions and irregular forms. For example, we can set up programs to define certain words like "and", and "or" as "demons", which cause an interrupt in the parsing process whenever they are encountered in the normal left-to-right order, in order to run a special program for conjoined structures. Idioms can also be handled using this "interrupt" concept. In fact, the process can be interrupted at any point in the sentence, and any other computations (either semantic or syntactic) can be performed before going on. These may themselves do bits of parsing, or they may change the course the basic program will take after they are done.

It is paradoxical that linguistic workers familiar with computers have generally not appreciated the importance of the "control" aspect of programming, and have not used the process-describing potentialities of programming for their parsing theories. They have instead restricted themselves to the narrowest kinds of rules and transformations -- as though a programmer were to stick to such simple models as Turing machines or Post productions. Designers of computer languages

today show this same tendency! See Minsky's remark in his Turing lecture (Minsky 1970). Our parser uses semantic guidance at all points, looking for a meaningful parsing of the sentence rather than trying all of the syntactic possibilities. Section 2.2 describes PROGRAMMAR in detail, and 2.3 gives a sample grammar for English. Section 2.4 explains programming details, and shows how the special features of the language are actually used to handle specific linguistic problems.

2.2 A Description of PROGRAMMAR

2.2.1 Grammar and Computers

In order to explain the features of PROGRAMMAR, we will summarize some of the principles of grammar used in computer language processing. The basic form of most grammars is a list (ordered or unordered) of "replacement rules," which represent a process of sentence generation. Each rule states that a certain string of symbols (its left side) can be replaced by a different set of symbols (its right side). These symbols include both the actual symbols of the language (called terminal symbols) and additional "non-terminal" symbols. One non-terminal symbol is designated as a starting symbol, and a string of terminal symbols is a sentence if and only if it can be derived from the starting symbol through successive application of the rules. For example we can write Grammar 1:

```

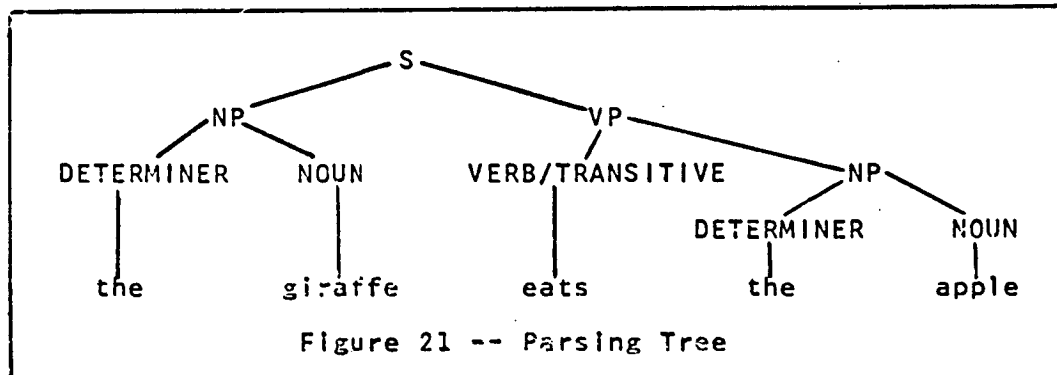
1.1 S -> NP VP
1.2 NP -> DETERMINER NOUN
1.3 VP -> VERB/INTRANSITIVE
1.4 VP -> VERB/TRANSITIVE NP
1.5 DETERMINER -> the
1.6 NOUN -> giraffe
1.7 NOUN -> apple
1.8 VERB/INTRANSITIVE -> dreams
1.9 VERB/TRANSITIVE -> eats

```

Figure 20 -- GRAMMAR 1

By starting with S and applying the list of rules (1.1 1.2 1.5 1.6 1.4 1.2 1.7 1.5 1.9), we get the sentence "The giraffe eats the apple." Several things are noteworthy here. This is an

unordered set of rules. Each rule can be applied any number of times at any point in the derivation where the symbol appears. In addition, each rule is optional. We could just as well have reversed the applications of 1.6 and 1.7 to get "The apple eats the giraffe.", or have used 1.3 and 1.8 to get "The giraffe dreams." This type of derivation can be represented graphically as:



We will call this the parsing tree for the sentence, and use the usual terminology for trees (node, subtree, daughter, parent, etc.). In addition we will use the linguistic terms "phrase" and "constituent" interchangeably to refer to a subtree. This tree represents the "immediate constituent" structure of the sentence. The PROGRAMMAR language is a general parsing system which, although oriented toward systemic grammar, can be used to parse grammars based on other theories. In describing PROGRAMMAR we have used a more conventional set of notations and analysis of English in order to make the description independent of the work presented in later sections.

2.2.2 Context-free and Context-sensitive Grammars

Grammar 1 is an example of what is called a context-free grammar. The left side of each rule consists of a single symbol, and the indicated replacement can occur whenever that symbol is encountered. There are a great number of different forms of grammar which can be shown to be equivalent to this one, in that they can characterize the same languages. It has been pointed out that they are not theoretically capable of expressing the rules of English, to produce such sentences as, "John, Sidney, and Chan ordered an eggroll, a ham sandwich, and a bagel respectively." Much more important, even though they could theoretically handle the bulk of the English language, they cannot do this at all efficiently. Consider the simple problem of subject-verb agreement. We would like a grammar which generates "The giraffe dreams." and "The giraffes dream.", but not "The giraffe dream." or "The giraffes dreams.". In a context-free grammar, we can do this by introducing two starting symbols, S/PL and S/SG for plural and singular respectively, then duplicating each rule to match. For example, we would have:

```

1.1.1 S/PL -> NG/PL VP/PL
1.1.2 S/SG -> NG/SG VP/SG
1.2.1 NG/PL -> DETERMINER NOUN/PL
1.2.2 NG/SG -> DETERMINER NOUN/SG
...
```

```

1.6.1 NOUN/PL -> giraffes
1.6.2 NOUN/SG -> giraffe
```

etc.

If we then wish to handle the difference between "I am", "he is", etc. we must introduce an entire new set of symbols for first-person. This sort of duplication propagates multiplicatively through the grammar, and arises in all sorts of cases. For example, a question and the corresponding statement will have much in common concerning their subjects, objects verbs, etc., but in a context-free grammar, they will in general be expanded through two entirely different sets of symbols.

One way to avoid this problem is to use context-sensitive rules. In these, the left side may include several symbols, and the replacement occurs when that combination of symbols occurs in the string being generated.

2.2.3 Systemic Grammar

We can add power to our grammar with context-sensitive rules which, for example, in expanding the symbol VERB/INTRANSITIVE, look to the preceding symbol to decide whether it is singular or plural. By using such context-sensitive rules, we can characterize any language whose sentences can be listed by a deterministic (possibly neverending) process. (i.e. they have the power of a turing machine). There is however a problem in implementing these rules. In any but the simplest cases, the context will not be as obvious as in the simple example given. The choice of replacements will not depend on a single word, but may depend in a complex way on the entire structure of the sentence. Such dependencies cannot be expressed in our simple rule format, and new types of rules must be developed. Transformational grammar solves this by breaking the generation process down into the context-free base grammar which produces "deep structure" and a set of transformations which then operate on this structure to produce the actual "surface structure" of the grammatical sentence. We will not go into the details of transformational grammar, but one basic idea is this separation of the complex aspects of language into a separate transformational phase of the generation process.

Systemic grammar introduces context in a more unified way into the immediate-constituent generation rules. This is done by introducing "features" associated with constituents at every

level of the parsing tree. A rule of the grammar may depend, for example, on whether a particular clause is transitive or intransitive. In the examples "Fred found a frog.", "A frog was found by Fred.", and "What did Fred find?", all are transitive, but the outward forms are quite different. A context-sensitive rule which checked for this feature directly in the string being generated would have to be quite complex. Instead, we can allow each symbol to have additional subscripts, or features which control its expansion. In a way, this is like the separation of the symbol NP into NP/PL and NP/SG in our augmented context-free grammar. But it is not necessary to develop whole new sets of symbols with a set of expansions for each. A symbol such as CLAUSE may be associated with a whole set of features (such as TRANSITIVE, QUESTION, SUBJUNCTIVE, OBJECT-QUESTION, etc.) but there is a single set of rules for expanding CLAUSE. These rules may at various points depend on the set of features present.

The power of systemic grammar rests on the observation that the context-dependency of natural language is centered around clearly defined and highly structured sets of features, so through their use a great deal of complexity can be handled very economically. More important for our purposes, there is a high correlation between these features and the semantic interpretation of the constituents which exhibit them. They cannot be put in a one-to-one correspondence with semantic properties of the phrases in which they appear, but are a tremendous aid to interpretation.

A parsing of a sentence in a systemic grammar might look very much like a context-free parsing tree, except that to each node would be attached a number of features. These features are

not random combinations of facts about the constituent, but are a part of a carefully worked out analysis of a language in terms of its "systems". The features are organized in a network, with clearly organized dependencies. For example, the features IMPERATIVE (command) and QUESTION are mutually exclusive in a clause, as are the features YES-NO (yes-no question like "Did he go?") and WH- question (like "Who went?"). In addition, the second choice can be made only if the choice QUESTION was made in the first set. A set of mutually exclusive features is called a "system", and the set of other features which must be present for the choice to be possible is called the "entry condition" for that system. This is discussed in detail in section 2.3.

Another basic concept of systemic grammar is that of the rank of a constituent. Rather than having a plethora of different non-terminal symbols, each expanding a constituent in a slightly different way, there are only a few basic "units", each having the possibility of a number of different features, chosen from the "system network" for that unit. In an analysis of English, three basic units seem to explain the structure: the CLAUSE, the GROUP, and the WORD. In general, clauses are made up of groups, and groups made up of words. However, through "rankshift", clauses or groups can serve as constituents of other clauses or groups. Thus, in the sentence "Sarah saw the student sawing logs." "the student sawing logs" is a NOUN GROUP

with the CLAUSE "sawing logs" as a constituent (a modifier of "student").

The constituents "who", "three days", "some of the men on the board of directors," and "anyone who doesn't understand me" are all noun groups, exhibiting different features. This means that a PROGRAMMAR grammar will have only a few programs, one to deal with each of the basic units. Our current grammar of English has programs for the units CLAUSE, NOUN GROUP, VERB GROUP, PREPOSITION GROUP, and ADJECTIVE GROUP.

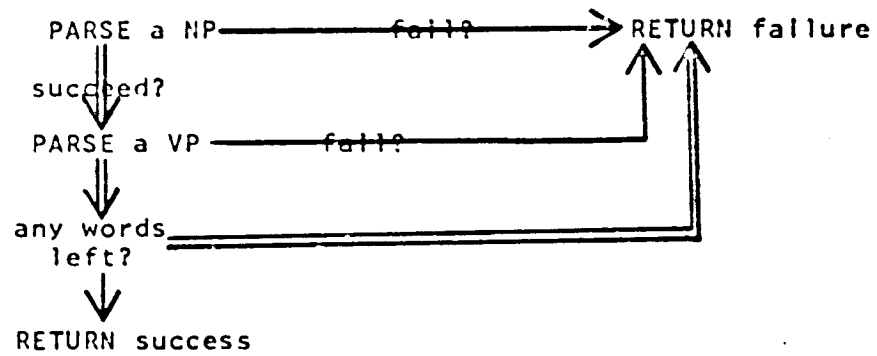
2.2.4 Grammars as Programs

Earlier we pointed out that a complete generative description of a language can be in the form of a program for parsing it. For simple grammars, there is a close correspondence between the parsing program and the usual generation rules.

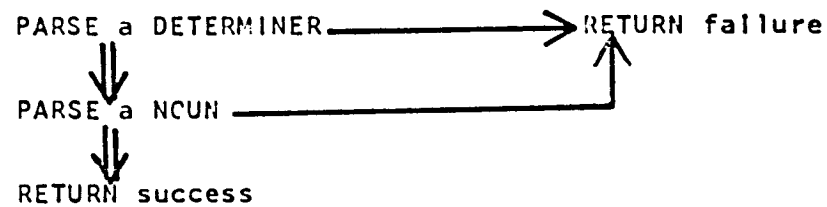
We can think of a grammar as a set of instructions for parsing a sentence in the language. A rule like: NP -> DETERMINER NOUN can be interpreted as the instruction "If you want to find a NP, look for a DETERMINER followed by a NOUN." Grammar 1 could be diagrammed as shown in Figure 22.

The basic function used is PARSE, a function which tries to add a constituent of the specified type to the parsing tree. If the type has been defined as a PROGRAMMAR program, PARSE activates the program for that unit, giving it as input the part of the sentence yet to be parsed and (optionally) a list of initial features. If no definition exists, PARSE interprets its arguments as a list of features which must be found in the dictionary definition of the next word in the sentence. If so, it attaches a node for that word, and removes it from the remainder of the sentence. If not, it fails. If a PROGRAMMAR program has been called and succeeds, the new node is attached to the parsing tree. If it fails, the tree is left unchanged.

DEFINE program SENTENCE



DEFINE program NP



DEFINE program VP

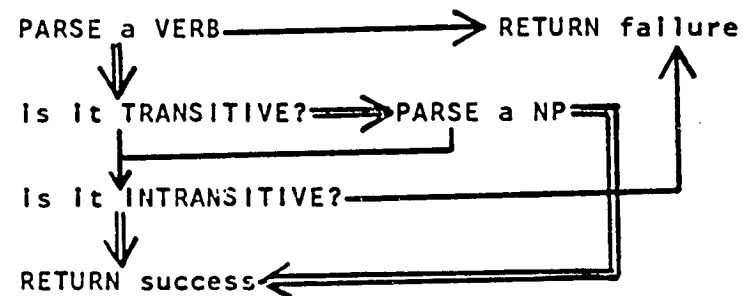


Figure 22 --Simple Parsing Program

2.2.5 The Form of PROGRAMMAR Grammars

Written in PROGRAMMAR, the programs would look like:

```

2.1 (PDEFINE SENTENCE
2.2 (((PARSE NP) NIL FAIL)
2.3 ((PARSE VP) FAIL FAIL RETURN)))

2.4 (PDEFINE NP
2.5 (((PARSE DETERMINER) NIL FAIL)
2.6 ((PARSE NOUN) RETURN FAIL)))

2.7 (PDEFINE VP
2.8 (((PARSE VERB) NIL FAIL)
2.9 ((ISQ H TRANSITIVE) NIL INTRANS )
2.10 ((PARSE NP) RETURN NIL)
2.11 INTRANS
2.12 ((ISQ H INTRANSITIVE) RETURN FAIL)))

Rules 1.6 to 1.9 would have the form:

2.13 (DEFPROP GIRAFFE (NOUN) WORD)
2.14 (DEFPROP DREAM (VERB INTRANSITIVE) WORD)
etc.

```

Figure 23 -- Grammar 2

This example illustrates some of the basic features of PROGRAMMAR. First it is embedded in LISP, and much of its syntax is LISP syntax. Units, such as SENTENCE are defined as PROGRAMMAR programs of no arguments. Each tries to parse the string of words left to be parsed in the sentence. The exact form of this input string is described in section 2.4.8. The value of (PARSE SENTENCE) will be a list structure corresponding to the parsing tree for the complete sentence.

Each time a call is made to the function PARSE, the system begins to build a new node on the tree. Since PROGRAMMAR programs can call each other recursively, it is necessary to

keep a pushdown list of nodes which are not yet completed (i.e. the entire rightmost branch of the tree). These are all called "active" nodes, and the one formed by the most recent call to PARSE is called the "currently active node".

We can examine our sample program to see the basic operation of the language. Whenever a PROGRAMMAR program is called directly by the user, a node of the tree structure is set up, and a set of special variables are bound (see section 2.4.9). The lines of the program are then executed in sequence, as in a LISP PROG, except when they have the special form of a BRANCH statement (a list whose first member (the CONDITION) is non-atomic, and which has either 2 or 3 other members, called DIRECTIONS). Line 2.3 of GRAMMAR 2 is a three-direction branch, and all the other executable lines of the program are two-direction branches.

When a branch statement is encountered, the condition is evaluated, and branching depends on its value. In a two-direction branch, the first direction is taken if it evaluates to non-NIL, the second direction if it is NIL. In a three-direction branch, the first direction is taken only if the condition is non-NIL, and there is more of the sentence to be parsed. If no more of the sentence remains, and the condition evaluates non-NIL, the third direction is taken.

The directions can be of three types. First, there are three reserved words, NIL, RETURN, and FAIL. A direction of NIL

sends evaluation to the next statement in the program. FAIL causes the program to return NIL after restoring the sentence and the parsing tree to their state before that program was called. RETURN causes the program to attach the currently active node to the completed parsing tree and return the subtree below that node as its value.

If the direction is any other atom, it acts as a GO statement, transferring evaluation to the statement immediately following the occurrence of that atom as a tag. For example, if a failure occurs in line 2.9, evaluation continues with line 2.12. If the direction is non-atomic, the result is the same as a FAIL, but the direction is put on a special failure message list, so the calling program can see the reason for failure. DIRECTIONS can also be used in the function GOCOND. The statement (GOCOND TAG1 TAG2) causes the program to go to TAG1 if there are words left to be parsed, and to TAG2 otherwise.

Looking at the programs, we see that SENTENCE will succeed only if it first finds a NP, then finds a VP which uses up the rest of the sentence. In the program VP, we see that the first branch statement checks to see whether the next word is a verb. If so, it removes it from the remaining sentence, and goes on. If not, VP fails. The second statement uses the PROGRAMMAR function ISQ, one of the functions used for checking features. (ISQ A B) checks to see whether the node or word pointed to by A has the feature B. H is one of a number of special variables

used to hold information associated with a node of the parsing tree. (see section 2.4.9) It points to the last word or constituent parsed by that program. Thus the condition (ISQ H TRANSITIVE) succeeds only if the verb just found by PARSE has the feature TRANSITIVE. If so, the direction NIL sends it on to the next statement to look for a NP, and if it finds one it returns success. If either no such NP is found or the verb is not TRANSITIVE, control goes to the tag INTRANS, and if the verb is INTRANSITIVE, the program VP succeeds. Note that a verb can have both the features INTRANSITIVE and TRANSITIVE, and the parsing will then depend on whether or not an object NP is found.

2.2.6 Context-Sensitive Aspects

So far, we have done little to go beyond a context-free grammar. How, for example, can we handle agreement? One way to do this would be for the VP program to look back in the sentence for the subject, and check its agreement with the verb before going on. We need a way to climb around on the parsing tree, looking at its structure. In PROGRAMMAR, this is done with the pointer PT and the moving function *.

Whenever the function * is called, its arguments form a list of instructions for moving PT from its present position. These instructions can be quite general, saying things like "Move left until you find a unit with feature X, then up until you find a CLAUSE, then down to its last constituent, and left until you find a unit meeting the arbitrary condition Y." The instruction list contains non-atomic CONDITIONS and atomic INSTRUCTIONS. The instructions are taken in order, and when a condition is encountered, the preceding instruction is evaluated repeatedly until the condition is satisfied. If the condition is of the form (ATOM), it is satisfied only if the node pointed to by PT has the feature ATOM. Any other condition is evaluated by LISP, and is satisfied if it returns a non-NIL value. Section 2.4.10 lists the instructions for *.

For example, evaluating (* C U) will set the pointer to the parent of the currently active node. (The mnemonics are: Current, Up) The call (* C DLC PV (NP)) will start at the

current node, move down to the rightmost completed node (i.e. not currently active) then move left until it finds a node with the feature NP. (Down-Last-Completed, PreVIOUS). If * succeeds, it returns the new value of PT and leaves PT set to that value. If it fails at any point in the list, because the existing tree structure makes a command impossible, or because a condition cannot be satisfied, PT is left at its original position, and * returns NIL.

We can now add another branch statement to the VP program in section 2.2.5 between lines 2.8 and 2.9 as follows:

```

2.8.1 ((OR(AND(ISQ(* C PV DLC)SINGULAR)(ISQ H SINGULAR))
2.8.2      (AND(ISQ PT PLURAL)(ISQ H PLURAL)))
2.8.3  NIL (AGREEMENT))

```

This is an example of a branch statement with an error message. It moves the pointer from the currently active node (the VP) to the previous node (the NP) and down to its last constituent (the noun). It then checks to see whether this shares the feature SINGULAR with the last constituent parsed by VP (the verb). If not it checks to see whether they share the feature PLURAL. Notice that once PT has been set by *, it remains at that position. If agreement is found, evaluation continues as before with line 2.9. If not, the program VP fails with the message (AGREEMENT).

So far we have not made much use of features, except on words. As the grammar gets more complex, they become much more important. As a simple example, we may wish to augment our

grammar to accept the noun groups "these fish," "this fish," "the giraffes," and "the giraffe," but not "these giraffe," or "this giraffes." We can no longer check a single word for agreement, since "fish" gives no clue to number in the first two, while "the" gives no clue in the third and fourth. Number is a feature of the entire noun group, and we must interpret it in some cases from the form of the noun, and in others from the form of the determiner.

We can rewrite our programs to handle this complexity as shown in Grammar 3:

```

3.1 (PDEFINE SENTENCE
3.2 (((PARSE NP)NIL FAIL)
3.3 ((PARSE VP) FAIL FAIL RETURN)))

3.4 (PDEFINE NP
3.5 (((AND(PARSE DETERMINER)(FQ DETERMINED))NIL NIL FAIL)
3.6 ((PARSE NOUN)NIL FAIL)
3.7 ((CQ DETERMINED)DET NIL)
3.8 ((AND(* H)(TRNSF (QUOTE(SINGULAR PLURAL))))RETURN FAIL)
3.9 DET
3.10 ((TRNSF (MEET(FE(* H PV (DETERMINER)))
3.11 (QUOTE(SINGULAR PLURAL))))
3.12 RETURN
3.13 FAIL)))

3.14 (PDEFINE VP
3.15 (((PARSE VERB)NIL FAIL)
3.16 ((MEET(FE H)(FE(* C PV (NP)))(QUOTE(SINGULAR PLURAL)))
3.17 NIL
3.18 (AGREEMENT))
3.19 ((ISQ H TRANSITIVE)NIL INTRANS)
3.20 ((PARSE NP)RETURN NIL)
3.21 ((ISQ H INTRANSITIVE)RETURN FAIL)))

```

Figure 24 -- Grammar 3

We have used the PROGRAMMAR functions FQ and TRNSF, which attach features to constituents. The effect of evaluating (FQ A) is to add the feature A to the list of features for the currently active node of the parsing tree. TRNSF is used to transfer features from the pointer to the currently active node. Its argument is a list of features to be looked for. For example, line 3.8 looks for the features SINGULAR and PLURAL in the last constituent parsed (the NOUN), and adds whichever ones it finds to the currently active node. The branch statement beginning with line 3.10 is more complex. The function * finds the DETERMINER of the NP being parsed. The function FE finds the list of features of this node, and the function MEET intersects this with the list of features (SINGULAR PLURAL). This intersection is then the set of allowable features to be transferred to the NP node from the NOUN. Therefore if there is no agreement between the NOUN and the DETERMINER, TRNSF fails to find any features to transfer, and the resulting failure causes the rejection of such phrases as "these giraffe."

In line 3.7 we use the function CQ which checks for features on the current node. (CQ DETERMINED) will be non-NIL only if the current node has the feature DETERMINED. (i.e. it was put there in line 3.5) Therefore, a noun group with a determiner is marked with the feature DETERMINED, and is also given features corresponding to the intersection of the number features associated with the determiner if there is one, and the

noun. Notice that this grammar can accept noun groups without determiners, as in "Giraffes eat apples." since line 3.5 fails only if a DETERMINER is found and there are no more words in the sentence.

In conjunction with the change to the NP program, the VP program must be modified to check with the NP for agreement. The branch statement beginning on Line 3.16 does this by making sure there is a number feature common to both the subject and the verb.

This brief description explains some of the basic features of PROGRAMMAR. In a simple grammar, their importance is not obvious, and indeed there seem to be easier ways to achieve the same effect. As grammars become more complex, the special aspects of PROGRAMMAR become more and more important. The flexibility of writing a grammar as a program is needed both to handle the complexities of English syntax, and to combine the semantic analysis of language with the syntactic analysis in an intimate way. Section 2.3 describes a fairly complex grammar of English, and section 4.2 describes the way it is integrated with the semantic programs. A number of the other features and details of PROGRAMMAR are described in Section 2.4.

2.2.7 Ambiguity and Understanding

Readers familiar with parsing systems may by now have wondered about the problem of ambiguity. As explained, a PROGRAMMAR program tries to find a possible parsing for a sentence, and as soon as it succeeds, it returns its answer. This is not a defect of the system, but an active part of the concept of language for which it was designed. The language process is not segmented into the operation of a parser, followed by the operation of a semantic interpreter. Rather, the process is unified, with the results of semantic interpretation being used to guide the parsing. This is very difficult in other forms of grammar, with their restricted types of context-dependence. But it is straightforward to implement in PROGRAMMAR. For example, the last statement in a program for NP may be a call to a noun-phrase semantic interpreter. If it is impossible to interpret the phrase as it is found, the parsing is immediately redirected.

The way of treating ambiguity is not through listing all 1243 possible interpretations of a sentence, but in being intelligent in looking for the first one, and being even more intelligent in looking for the next one if that fails. There is no automatic backup mechanism in PROGRAMMAR, because blind automatic backup is tremendously inefficient. A good PROGRAMMAR program will check itself when a failure occurs, and based on the structures it has seen and the reasons for the failure, it

will decide specifically what should be tried next. This is the reason for internal failure-messages, and there are facilities for performing the specific backup steps necessary. (See section 2.4.5)

As a concrete example, we might have the sentence "I rode down the street in a car." At a certain point in the parsing, the NP program may come up with the constituent "the street in a car". Before going on, the semantic analyzer will reject the phrase "in a car" as a possible modifier of "street", and the program will attach it instead as a modifier of the action represented by the sentence. Since the semantic system is a part of a complete deductive understander, with a definite world-model, the semantic evaluation which guides parsing can include both general knowledge (cars don't contain streets) and specific knowledge (Melvin owns a red car, for example). Humans take advantage of this sort of knowledge in their understanding of language, and it has been pointed out by a number of linguists and computer scientists that good computer handling of language will not be possible unless computers can do so as well.

Few sentences seem ambiguous to humans when first read. They are guided by an understanding of what is said to pick a single parsing and a very few different meanings. By using this same knowledge to guide its parsing, a computer understanding system can take advantage of the same technique to parse meaningful sentences quickly and efficiently.

We must be careful to distinguish between grammatical and semantic ambiguity. Although we want to choose a single parsing

without considering the alternatives simultaneously, we want to handle semantic ambiguity very differently. There may be several interpretations of a sentence which are all more or less meaningful, and the choice between them will depend on a complex evaluation of our knowledge of the world, of the knowledge the person speaking has of the world, and of what has been said recently.

2.2.8 Summary

In understanding the reason for developing PROGRAMMAR, several factors are important. The first is that only through the flexibility of expressing a grammar as a program can we introduce the type of intelligence necessary for complete language understanding. PROGRAMMAR is able to take into account the fact that language is structured in order to convey meaning, and that our parsing of sentences depends intimately on our understanding of that meaning. PROGRAMMAR can take advantage of this to deal more efficiently with natural language than a general rule-based system, whether context-free or transformational. More important, the analysis returned by PROGRAMMAR is designed to serve as a part of a total understanding process, and to lend itself directly to semantic interpretation. This was one reason for selecting systemic grammar, and has guided much of the design of the system. The exact way in which semantic interpretation can be done, and the reasons why a systemic analysis is important will be discussed in sections 4.1 and 4.2.

2.3 A Grammar of English

2.3.1 About the Grammar

This section describes the grammar of English used by our system. It is based on the principles of systemic grammar (<Halliday 1961, 1966a, 1966b, 1967>), and emphasizes the analysis of the significant features exhibited by linguistic structures, rather than their detailed form. Instead of giving a set of grammar rules or algorithms, this section will try to convey an impressionistic overview of English grammar, giving examples of the most important features. Section 2.3.11 follows the parser through two sample sentences, showing how the grammar is used. The actual PROGRAMMAR grammar contains the details, and is available on request. A few particularly interesting parts of the detailed grammar are described in section 2.4. Appendix A is a glossary of features with references to the pages on which they are illustrated.

Before beginning, several warnings are in order. First, this is not by any means a complete grammar of English. The task of codifying an entire language in any formalism is so large that it would be folly to try in the course of a single research project. Our goal was to cover a large enough portion of English syntax so that a user could converse comfortably with the system about its toy-block world. There are whole areas of syntax which are involved with conveying information of types not included in this narrow field (such as the emotional

reaction, mood, and emphasis of the speaker). These are not handled at all, and even within the toy-block world, there are numerous sentences and constructions which the grammar is not yet equipped to handle. It will be of interest to see whether the basic structure of the syntactic theory is flexible enough to add the great amount of complexity which could be included in a more complete grammar.

Second, the grammatical theory is used in a very impure way. The main consideration was to produce a working grammar which could serve in a language-understanding program. The demands of practicality often overrode more theoretical criteria, and the resulting grammar is not very "pretty". This is especially true since it has evolved in a continuous process of writing and debugging, and has not yet undergone the "polishing" which removes the traces of its earlier stages of development.

Demands of time made it impossible to coordinate the writing of the grammar with other current versions of systemic grammar, so the analysis is non-standard, often disagreeing with Halliday's analysis or other more complete versions. Some differences are simply notational (using different names for the same thing), others are intentional simplifications (Halliday's analysis is much more complete), and some represent actual theoretical differences (for example, our analysis of the transitivity system puts much of the structure into the semantic

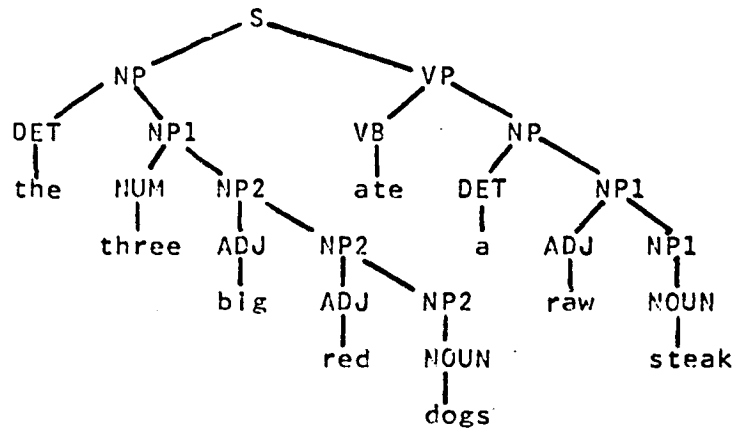
rather than syntactic rules, while Halliday's is more purely syntactic.). We will not describe the differences in detail, since this is not a proposal for a specific version of English grammar. It is instead a proposal for a way of looking at language, and at English, pointing out some of the interesting features.

2.3.2 Units, Rank, and Features

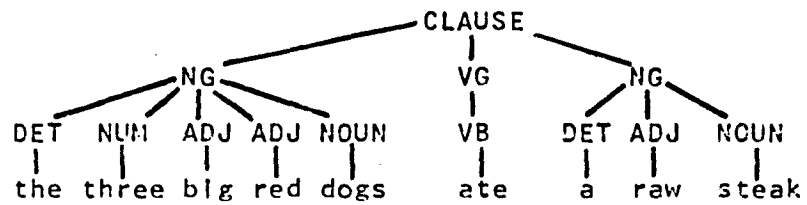
We will begin by describing some of the basic concepts of systemic grammar, before giving details of their use in our analysis. Some of the description is a repetition of material in Section 2.1. In that section we needed to give enough explanation of systemic grammar to explain PROGRAMMAP. Here we give a more thorough explanation of its details.

The first is the notion of syntactic units in analyzing the constituent structure of a sentence (the way it is built up out of smaller parts). If we look at other forms of grammar, we see that syntactic structures are usually represented as a binary tree, with many levels of branching and few branches at any node. The tree is not organized into "groupings" of phrases which are used for conveying different parts of the meaning. For example, the sentence "The three big red dogs ate a raw steak." would be parsed with something like the first tree in Figure 25.

Systemic grammar pays more attention to the way language is organized into units, each of which has a special role in conveying meaning. In English we can distinguish three basic ranks of units, the CLAUSE, the GROUP, and the WORD. There are several types of groups: NOUN GROUP (NG), VERB GROUP (VG) PREPOSITION GROUP (PREPG) and ADJECTIVE GROUP (ADJG). In a systemic grammar, the same sentence might be viewed as having the second structure in Figure 25.



Tree 1



Tree 2

Figure 25 - Parsing Trees

In this analysis, the WORD is the basic building block. There are word classes like "adjective", "noun", "verb", and each word is an integral unit -- it is not chopped into hypothetical bits (like analyzing "dogs" as being composed of "dog" and "-s" or "dog" and "plural"). Instead we view each word as exhibiting features. The word "dogs" is the same basic vocabulary item as "dog", but has the feature "plural" instead of "singular". The words "took", "take", "taken", "taking", etc., are all the same basic word, but with differing features such as "past participle" (EN), "infinitive" (INF), "-ing" (ING), etc. When discussing features, we will use several notational conventions. Any word appearing in all upper-case letters, is the actual symbol used to represent a feature in our grammar and semantic programs. A feature name enclosed in quotes is an English version which is more informative. Usually the program version is an abbreviation of the English version, and sometimes we will indicate this by typing the letters of the abbreviation in upper-case, and the rest in lower-case. Thus if "determiner" is abbreviated as DET, we may write DETerminer. We may even write things like QuANTIFIER. When we want to be more careful, we will write "quantifier" (QNTFR).

The next larger unit than the WORD is the GROUP, of which there are the four types mentioned above. Each one has a particular function in conveying meaning. Noun groups (NG) describe objects, verb groups (VG) carry complex messages about

the time and modal (logical) status of an event or relationship, preposition groups (PREPG) describe simple relationships, while adjective groups (ADJG) convey other kinds of relationships and descriptions of objects. These semantic functions are described in more detail in section 4.2.

Each GROUP can have "slots" for the words of which it is composed. For example, a NG has slots for "determiner" (DET), "numbers" (NUM), "adjectives" (ADJ), "classifiers" (CLASF), and a NOUN. Each group can also exhibit features, just as a word can. A NG can be "singular" (NS) or "plural" (NPL), "definite" (DEF) as in "the three dogs" or "indefinite" (INDEF) as in "a steak", and so forth. A VG can be "negative" (NEG) or not, can be MODAL (as in "could have seen"), and it has a tense. (See Section 2.3.8 for an analysis of complicated tenses, such as "He would have been going to be fixing it.")

Finally, the top rank is the CLAUSE. We speak of clauses rather than sentences since the sentence is more a unit of discourse and semantics than a separate syntactic structure. It is either a single clause or a series of clauses joined together in a simple structure such as "A and B and...". We study these conjoining structures separately since they occur at all ranks, and there is no real need to have a separate syntactic unit for sentence.

The clause is the most complex and diverse unit of the language, and is used to express complex relationships and

events, involving time, place, manner and many other aspects of meaning. It can be a QUESTION, a DECLARATIVE, or an IMPERATIVE, it can be "passive" (PASV) or "active" (ACTV), it can be a YES-NO question or a WH- question (like "Why...?" or "Which...?").

Looking at our sample parsing tree, Tree 2 in Figure 25, we see that the clauses are made up of groups, which are in turn made up of words. However few sentences have this simple three-layer structure. Groups often contain other groups (for example, "the call of the wild" is a NG, which contains the PREPG "of the wild" which in turn contains the NG "the wild"). Clauses can be parts of other clauses (as in "Join the Navy to see the world."), and can be used as parts of groups in many different ways (for example, in the NG "the man who came to dinner" or the PREPG "by leaving the country".) This phenomenon is called rankshift, and is one of the basic principles of systemic grammar.

If the units can appear anywhere in the tree, what is the advantage of grouping constituents into "units" instead of having a detailed structure like the one shown in our first parsing tree? The answer is in the "features" we were noting above. Each unit has associated with it a set of features, which are of primary significance in conveying meaning. We mentioned that a clause could have features such as IMPERATIVE, DECLARATIVE, QUESTION, ACTV, PASV, YES-NO, and WH-. These are not unrelated observations we can make about a clause. They are

related by a definite logical structure. The choice between YES-NO and WH- is meaningless unless the clause is a QUESTION, but if it is a QUESTION, the choice must be made. Similarly, the choice between QUESTION, IMPERATIVE, and DECLARATIVE is mandatory for a MAJOR clause (one which could stand alone as a sentence), but is not possible for a "secondary" (SEC) clause, such as "the country which possesses the bomb." The choice between PASV (as in "the ball was attended by John",) and ACTV (as in "John attended the ball.") is on a totally different dimension, since it can be made regardless of which of these other features are present.

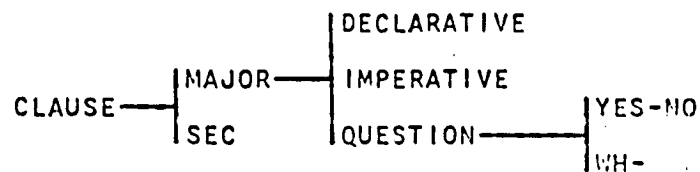
We can represent these logical relationships graphically using a few simple conventions. A set of mutually exclusive features (such as QUESTION, DECLARATIVE, and IMPERATIVE) is called a system, and is represented by connecting the features with a vertical bar:

QUESTION
DECLARATIVE
IMPERATIVE

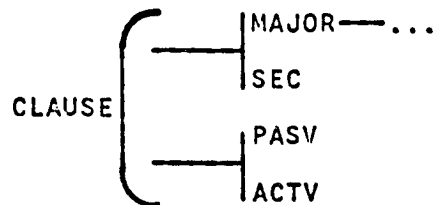
The vertical order is not important, since a system is a set of unordered features among which we will choose one. Each system has an entry condition which must be satisfied in order for the choice to be meaningful. This entry condition can be an arbitrary boolean condition on the presence of other features. The simplest case (and most common) is the presence of a single

other feature. For example, the system just depicted has the feature MAJOR as its entry condition, since only MAJOR clauses make the choice between DECLARATIVE, IMPERATIVE, and QUESTION.

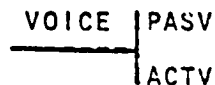
This simple entry condition is represented by a horizontal line, with the condition on the left of the system being entered. We can diagram some of our CLAUSE features as:



Often there are independent systems of choices sharing the same entry condition. For example, the choice between SEC and MAJOR and the choice between PASV and ACTV both depend directly on the presence of CLAUSE. This type of relationship will be indicated by a bracket in place of a vertical bar.



If we want to assign a name to a system (to talk about it), we can put the name above the line leading into it:

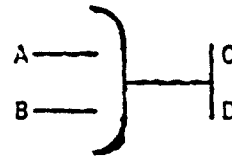


We can look at these notations as representing the logical operations of "or" and "and", and we can use them to represent more complex entry conditions. If the choice between the

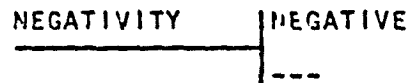
features C and D depends on the presence of either A or B, we draw;



and if the entry condition for the "C-D" system is the presence of both A and B, we write:



Finally, we can allow "unmarked" features, in cases where the choice is between the presence or absence of something of interest. We might have a system like;



In which the feature "non-negative" is not given a name, but is assumed unless the feature NEGATIVE is present.

We will explain our grammar by presenting the system networks for all three ranks -- CLAUSE, GROUP, and WORD, and giving examples of sentences exhibiting the features. We have not attempted to show all of the logical relationships in the networks -- our networks may indicate combinations of features which are actually not possible, and would need a more complex network to represent properly. We have chosen clarity over completeness whenever there was a conflict. In addition, we have represented "features" of units (i.e. descriptions of their

structure) and "functions" (descriptions of their use) in the same network. In a more theoretical presentation, it would be preferable to distinguish the two. The names chosen for features were arbitrary mnemonics invented as they were needed, and are neither as clear nor as systematic as they might be in a "cleaned up" version.

2.3.3 The CLAUSE

The structure exhibiting the greatest variety in English is the CLAUSE. It can express relationships and events involving time, place, manner, and other modifiers. Its structure indicates what parts of the sentence the speaker wants to emphasize, and can express various kinds of focus of attention and emotion. It determines the purpose of an utterance -- whether it is a question, command, or statement -- and is the basic unit which can stand alone. Other units can occur by themselves when their purpose is understood, as in answer to a question, but the clause is the primary unit of discourse.

The CLAUSE has several main ingredients and a number of optional ones. Except for special types of incomplete clauses, there is always a verb group, containing the verb, which indicates the basic event or relationship being expressed by the CLAUSE. Almost every CLAUSE contains a subject, except for IMPERATIVE (in which the semantic subject is understood to be the person being addressed), and embedded clauses in which the subject lies somewhere else in the syntactic structure. In addition to the subject, a CLAUSE may have various kinds of objects, which will be explained in detail later. It can take many types of modifiers (CLAUSES, GROUPS, and WORDS) which indicate time, place, manner, causality, and a variety of other aspects of meaning. One part of the CLAUSE system network is shown in Figure 26.

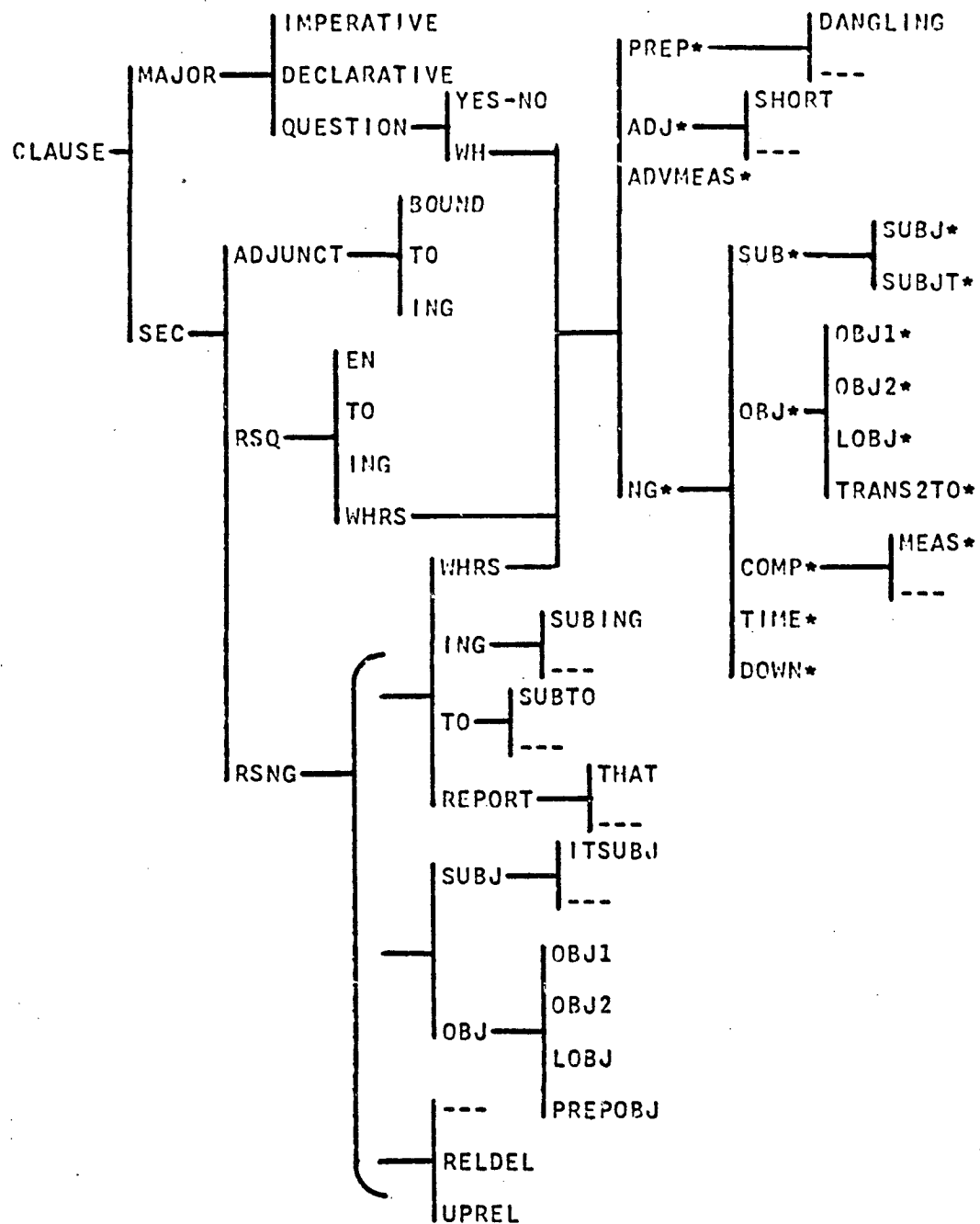


Figure 26 -- NETWORK 1

Beginning at the top of the network, we see a choice between MAJOR (a clause which could stand alone as a sentence) and "secondary" (SEC). A MAJOR clause is either an IMPERATIVE (a command), a DECLARATIVE, or a QUESTION. Questions are either YES-NO -- answerable by "yes" or "no", as in:

(s1) Did you like the show?

or WH- (Involving a question element like "when", "where", "which", "how", etc.). The choice of the WH- feature leads into a whole network of further choices, which are shared by QUESTION and two kinds of SECondary clauses we will discuss later. In order to share the network, we have used a simple notational trick -- the symbols contain a "*", and when they are being applied to a question, we replace the * with "Q", while when they are applied to relative clauses, we use "REL". For example, the feature "PREP*" in the network will be referred to as PREPQ when we find it in a question, but PREPREL when it is in a relative clause. This is due to the way the grammar evolved, and in later versions we will probably use only one name for these features. This complex of features is basically the choice of what element of the sentence is being questioned. English allows us to use almost any part of a clause as a request for information. For example, in a PREPQ, a prepositional group in the clause is used, as in:

(s2) With what did you erase it?

We more commonly find the preposition in a DANGLING position, as

In:

(s3) What did you erase it with?

We can tell by tracing back through Network 1 that sentence s3 has the features PREPQ, DANGLING, WH-, QUESTION, and MAJOR.

We can use a special question adverb to ask questions of time, place, and manner, as in:

(s4) Why did the chicken cross the road?

(s5) When were you born?

(s6) How will you tell her the news?

(s7) Where has my little dog gone?

These are all marked by the feature ADJQ. In discourse they can also appear in a short form (SHORT) in which the entire utterance is a single word, as in:

(s8) Why?

We can use the word "how" in connection with a measure adverb (like "fast") to ask an ADVMEASQ, like:

(s9) How fast can he run the mile?

The most flexible type of WH- question uses an entire noun group as the question element, using a special pronoun (like "what" or "who") or a determiner (like "which", or "how many") to indicate that it is the question element. These clauses have the feature NGQ, and they can be further divided according to the function of the NG in the clause. It can have any of the possible NG functions (these will be described more formally with regard to the next network). For example, it can be the subject, giving a SUBJQ, like:

(s10) Which hand holds the M and M's?

It can be the subject of a THERE clause (see below), giving us a SUBJTQ:

(s11) How many Puerto Ricans are there in Boston?

A complement is the second half of an "is" clause, like:

(s12) Her hair is red.

and it can be used to form a COMPQ:

(s13) What color was her hair?

or with a "measure" in a MEASQ:

(s14) How deep is the ocean?

The noun group can be an object, leading to the feature OBJQ, as in:

(s15) What do you want? or
(s16) Who did you give the book?

These are both OBJ1Q, since the first has only one object ("what"), and the second questions the first, rather than the second object ("who", instead of "the book"). We use the ordering of the DECLARATIVE form "You gave me the book". If this were reversed, we would have an OBJ2Q, like:

(s17) What did you give him?

If we use the word "to" to express the first object with a two object verb like "give", we can get a TRANST02Q, like:

(s18) To whom did you give the book? or
(s19) Who did you give the book to?

Sometimes a NG can be used to indicate the time in a clause, giving us a TIMEQ:

(s20) What day will the iceman come?

In a more complex style, we can embed the question element within an embedded clause, such as:

- (s21) Which car did your brother say that
he was expecting us to tell Jane to buy?

The NG "which car" is the question element, but is in fact the object of the clause "Jane to buy...", which is embedded several layers deep. This kind of NGQ is called DOWNQ. The role of the question element in the embedded clause can include any of those which we have been describing. For example it could be the object of a preposition, as in

- (s22) What state did you say Lincoln was born in?

Looking at the network for the features of SECondary clauses, we see three main types -- ADJUNCT, "Rank-Shifted Qualifier" (RSQ), and "Rank-Shifted to act as a Noun Group" (RSNG). ADJUNCT clauses are used as modifiers to other clauses, giving time references, causal relationships, and other similar information. We can use a BOUND clause containing a "binder" such as "before", "while", "because", "if", "so", "unless", etc., as in:

- (s23) While Nero fiddled, Rome burned.
(s24) If it rains, stay home.
(s25) Is the sky blue because it is cold?

To express manner and purpose, we use a T0 clause or an ING clause:

- (s26) He died to save us from our sins.
(s27) The bridge was built using primitive tools.

The RSQ clause is a constituent of a NG, following the noun

In the "qualifier" position (see Section 2.3.5 for a description of the positions in a NG). It is one of the most commonly used secondary clauses, and can be of four different types. Three of them are classified by the form of the verb group within the clause -- TO, ING, and EN (where we use "en" to represent a past participle, such as "broken"):

- (s28) the man to see about a job
- (s29) the piece holding the door on
- (s30) a face weathered by sun and wind

Notice that the noun being modified can have various roles in the clause. In examples 28 and 29, "piece" is the subject of "hold", while "man" is the object of "see". We could have said:

- (s31) the man to do the job

In which "man" is the subject of "do". Our semantic analysis sorts out these possibilities in determining the meaning of a secondary clause.

The fourth type of RSQ clause is related to WH- questions, and is called a WHRS. It uses a wh- element like "which" or "what", or a word like "that" to relate the clause to the noun it is modifying. The different ways it can use this "relating" element are very similar to the different possibilities for a question element in a WH- question, and in fact the two share part of the network. Here we use the letters REL to indicate we are talking about a relative clause, so the feature PREP* in Network 1 becomes PREPREL. In sentences (s2) through (s22), we illustrated the different types of WH- questions. We can show

parallel sentences for WHRS RSQ clauses. The following list shows some examples and the relevant feature names:

- (s32) the thing with which you erased it PREPREL
- (s33) the thing that you erased it with PREPREL DANGLING
- (s34) the reason why the chicken crossed the road RELADJ
- (s35) the day when you were born RELADJ
- (s36) the way we will tell her the news RELADJ
- (s37) the place my little dog has gone RELADJ
- (s38) the reason why RELADJ SHORTREL
- (s39) the hand which rocks the cradle SUBJREL
- (s40) the number of Puerto Ricans there are in Boston SUBJTREL
- (s41) the color her hair was last week COMPREL
- (s42) the depth the ocean will be MEASREL
- (s43) the information that you want OBJ1REL
- (s44) the man you gave the book OBJ1REL
- (s45) the book which you gave him OBJ2REL
- (s46) the man to whom you gave the book TRANSTO2REL
- (s47) the man you gave the book to TRANSTO2REL
- (s48) the day the iceman came TIMEREL
- (s49) the car your brother said he was expecting us to tell Jane to buy DOWNREL
- (s50) the state you said Lincoln was born in DOWNREL

Notice that in sentences 36, 37, 40, 41, 42, 44, 47, 48, 49, and 50, there is no relative word like "which" or "that". These could just as well all have been put in, but English gives us the option of omitting them. When they are absent, the CLAUSE is marked with the feature RELDEL.

Returning to our network, we see that there is one other type of basic clause, the RSNG. This is a clause which is rank-shifted to serve as a NG. It can function as a part of another clause, a preposition group, or an adjective group. There are four basic types. The first two are TO and ING, as in:

- (s51) I like to fly. TO
- (s52) Building houses is hard work. ING
- (s53) He got it by saving coupons. ING

Notice that in s51, the RSNG clause is the object (OBJ1), in s52 it is the subject (SUBJ), and in s53 it is the object of a preposition (PREPOBJ). We can have a separate subject within the TO and ING clauses, giving us the features SUBTO and SUBING:

- (s54) I wanted Ruth to lead the revolution. SUBTO
 (s55) They liked John's leading it. SUBING

The SUBING form takes its subject in the possessive.

In addition to ING and TO, we have the REPORT CLAUSE, which has the structure of an entire sentence, and is used as a participant in a relation about things like hearing, knowing, and saying:

- (s56) She heard that the other team had won.
 (s57) That she wasn't there surprised us.
 (s58) I knew he could do it.

The word "that" is used in s56 and s57 to mark the beginning of the REPORT CLAUSE, so they are assigned the feature THAT. The absence of "that" is left unmarked.

If the subject of a clause is in turn a RSNG clause, we may have trouble understanding it:

- (s59) That anyone who knew the combination could have opened the lock was obvious.

There is a special mechanism for rearranging the sentence by using the word "it", so that the complicated subject comes last:

- (s60) It was obvious that anyone who knew the combination could have opened the lock.

In this case, we say that the RSNG clause is serving as an

ITSUBJ. TO and ING clauses can do the same:

- (s61) It will be fun to see them again.
 (s62) It was dangerous going up without a parachute.

The final type of RSNG is the WHRS, which is almost identical to the WHRS RSQ described above. Rather than go through the details again, we will indicate how a few of our RSQ examples (sentences s32 to s50) can be converted, and will leave the reader to do the rest.

- (s63) I don't know what he did it with. PREPREL DANGLING
 (s64) Ask him when he was born. RELADJ
 (s65) He told me why. RELADJ SHORTREL
 (s66) It is amazing how many Puerto Ricans there are in Boston. SUBJTREL
 (s67) Only her hairdresser knows what color her hair was. COMPREL
 etc.

Let us examine one case more carefully:

- (s68) I knew which car your brother said that he was expecting us to tell Jane to buy.

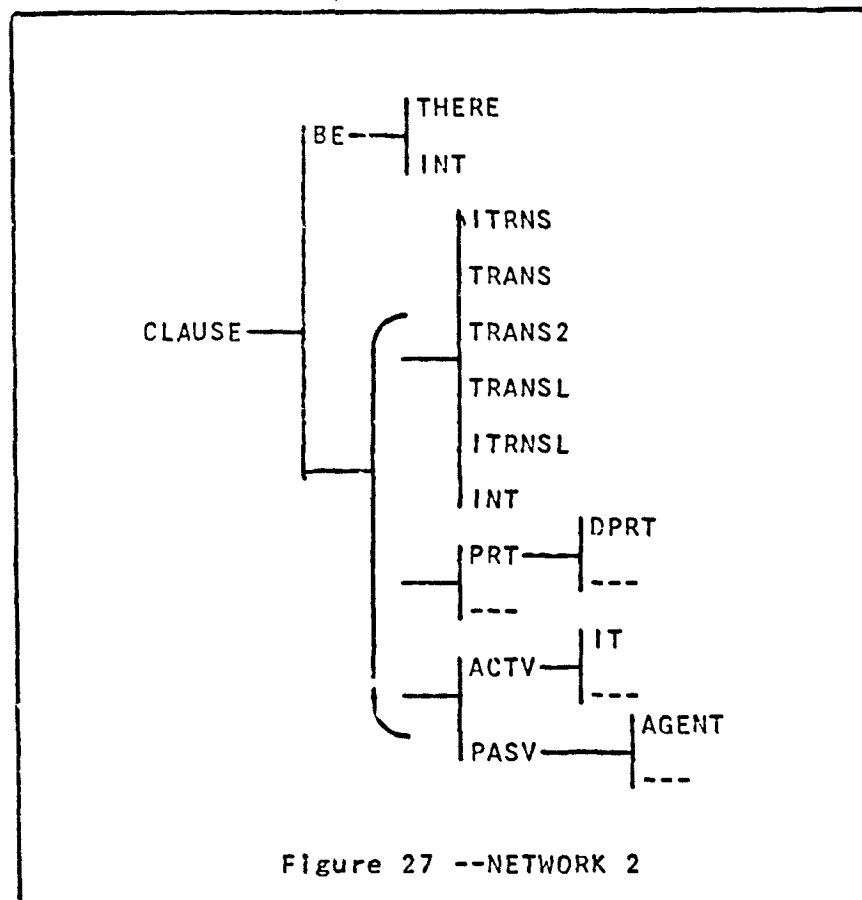
Here we have a DOWNREL clause, "which car....buy", serving as the object of the CLAUSE "I knew...". However, this means that somewhere below, there must be another clause with a slot into which the relative element can fit. In this case, it is the RSNG TO clause "Jane to buy", which is missing its object. This clause then has the feature UPREL, which indicates that its missing constituent is somewhere above in the structure. More specifically it is OBJ1UPREL.

Once this connection is found, the program might change the pointers in the structure to place the relative as the actual OBJ1 of the embedded clause structure. In the current grammar,

the pointers are left untouched, and special commands to the moving function * are used when the object is referenced by the semantic program.

2.3.4 Transitivity in the Clause

In addition to the systems we have already described, there is a TRANSITIVITY system for the CLAUSE, which describes the number and nature of its basic constituents. We mentioned earlier that a CLAUSE had such components as a subject and various objects. The transitivity system specifies these exactly. We have adopted a very surface-oriented notion of transitivity, in which we note the number and basic nature of the objects, but do not deal with their semantic roles, such as "range" or "beneficiary". Halliday's analysis (Halliday 1967) is somewhat different, as it includes aspects which we prefer to handle as part of the semantic analysis. Our simplified network is:



The first basic division is into clauses with the main verb "be", and those with other verbs. This is done since BE clauses have very different possibilities for conveying meaning, and they do not have the full range of syntactic choices open to other clauses. BE clauses are divided into two types -- THERE clauses, like:

(s69) There was an old woman who lived in a shoe.

and INTensive BE clauses:

(s70) War is hell.

A THERE CLAUSE has only a subject, marked SUBJT, while an INT

CLAUSE has a SUBJECT and a COMPLEMENT. The COMPLEMENT can be either a NG, as in s70 or:

(s71) He was an agent of the FBI.

or a PREPG:

(s72) The king was in the counting house.

or an ADJG:

(s73) Her strength was fantastic.

(s74) My daddy is stronger than yours.

Other clauses are divided according to the number and type of objects they have. A CLAUSE with no objects is Intransitive (ITRNS):

(s75) He is running.

With one object it is transitive (TRANS):

(s76) He runs a milling machine.

With two objects TRANS2:

(s77) I gave my love a cherry.

Some verbs are of a special type which use a location as a second object. One example is "put", as in:

(s78) Put the block on the table.

Note that this cannot be considered a TRANS with a modifier, as in:

(s79) He runs a milling machine in Chicago.

since the verb "put" demands that the location be given. We cannot say "Put the block." This type of CLAUSE is called TRANSL, and the location object is the LOBJ. The LOBJ can be a PREPG as in s78, or a special adverb, such as "there" or

"somewhere", as in:

- (s80) Where did you put it? or
 (s81) Put it there.

Some intransitive verbs also need a locational object for certain meanings, such as:

- (s82) The block is sitting on the table.

This is called ITRNSL.

Finally, there are INTensive clauses which are not BE clauses, but which have a COMPLEMENT, as in:

- (s83) He felt sick. and
 (s84) He made me sick.

We have not run into these with our simple subject matter, and a further analysis will be needed to handle them properly.

Any of the constituents we have been mentioning can be modified or deleted when these features interact with the features described in Network 1. For example in:

- (s85) the block which I told you to put on the table

the underlined CLAUSE is TRANSL, but its OBJ1 is missing since it is an UPREL.

English has a way of making up new words by combining a verb and a "particle" (PRT), producing a combination like "pick up", "turn on", "set off", or "drop out". These do not simply combine the meanings of the verb and particle, but there is a special meaning attached to the pair, which may be very different from either word in isolation. Our dictionary contains a table of such pairs, and the grammar programs use

them. A CLAUSE whose verb is a part of PRT pair has the feature PRT. The particle can appear either immediately after the word:

(s86) He threw away the plan.

or in a displaced position (marked by the feature DPRT):

(s87) He threw the plans away.

Regardless of whether there is a PRT or not, we have the choice between the features passive (PASV) and active (ACTV).

ACTV places the semantic subject first:

(s88) The President started the war.

while PASV puts the semantic object first:

(s89) The war was started by the President.

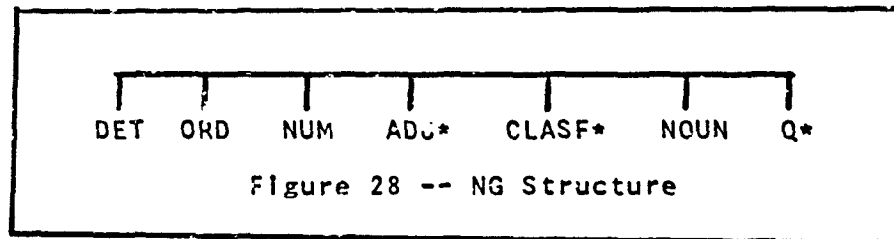
If there is a PREPG beginning with "by", it is interpreted as the semantic subject (as in s89), and the CLAUSE has the feature AGENT.

If the CLAUSE is active and its subject is a RSNG CLAUSE, we can use the IT form described earlier. This is marked by the feature IT, and its subject is marked ITSUBJ, as in sentences 60, 61, and 62.

2.3.5 Noun Groups

The best way to explain the syntax of the NOUN GROUP is to look at the "slot and filler" analysis, which describes the different components it can have. Some types of NG, such as those with pronouns and proper nouns, will not have this same construction, and they will be explained separately later.

We will diagram the typical NG structure, using a "*" to indicate that the same element can occur more than once. Most of these "slots" are optional, and may or may not be filled in any particular NG. The meanings of the different symbols are explained below.



The most important ingredient is the NOUN, which is almost always present (if it isn't, the NG is INCOMPLETE). It gives the basic information about the object or objects being referred to by the NG. Immediately preceding the NOUN, there are an arbitrary number of "classifiers" (CLASF). Examples of CLASF are:

- (s90) plant life
- (s91) water meter cover adjustment screw

Notice that the same class of words can serve as CLASF and NOUN -- in fact Halliday uses one word class (called NOUN), and

distinguishes between the functions of "head" and "classifier". We have separated the two because our dictionary gives the meaning of words according to their word class, and nouns often have a special meaning when used as a CLASF.

Preceding the CLASFs we have adjectives (ADJ), such as "big beautiful soft red..." We can distinguish adjectives from classifiers by the fact that adjectives can be used as the complement of a BE CLAUSE, but classifiers cannot. We can say "red hair", or "horse hair", or "That hair is red.", but we cannot say "That hair is horse.", since "horse" is a CLASF, not an ADJ. Adjectives can also take on the COMPARative and SUPERlative forms ("red, redder, and reddest"), while classifiers cannot ("horse, horser, and horsest"!!?).

Immediately following the NOUN we can have various qualifiers (Q), which can be a PREPG:

(s92) the man in the moon

or an ADJG:

(s93) a night darker than doom

or a CLAUSE RSQ:

(s94) the woman who conducts the orchestra

We have already discussed the many types of RSQ clauses. In later sections we will discuss the PREPG and ADJG types which can occur as qualifiers.

Finally, the first few elements in the NG work together to give its logical description -- whether it refers to a single

object, a class of objects, a group of objects, etc. The determiner (DET) is the normal start for a NG, and can be a word such as "a", or "that", or a possessive. It is followed by an "ordinal" (ORD). There is an infinite sequence of number ordinals ("first, second, third...") and a few others such as "last" and "next". These can be recognized since they are the only words that can appear between a DET like "the" and a number, as in:

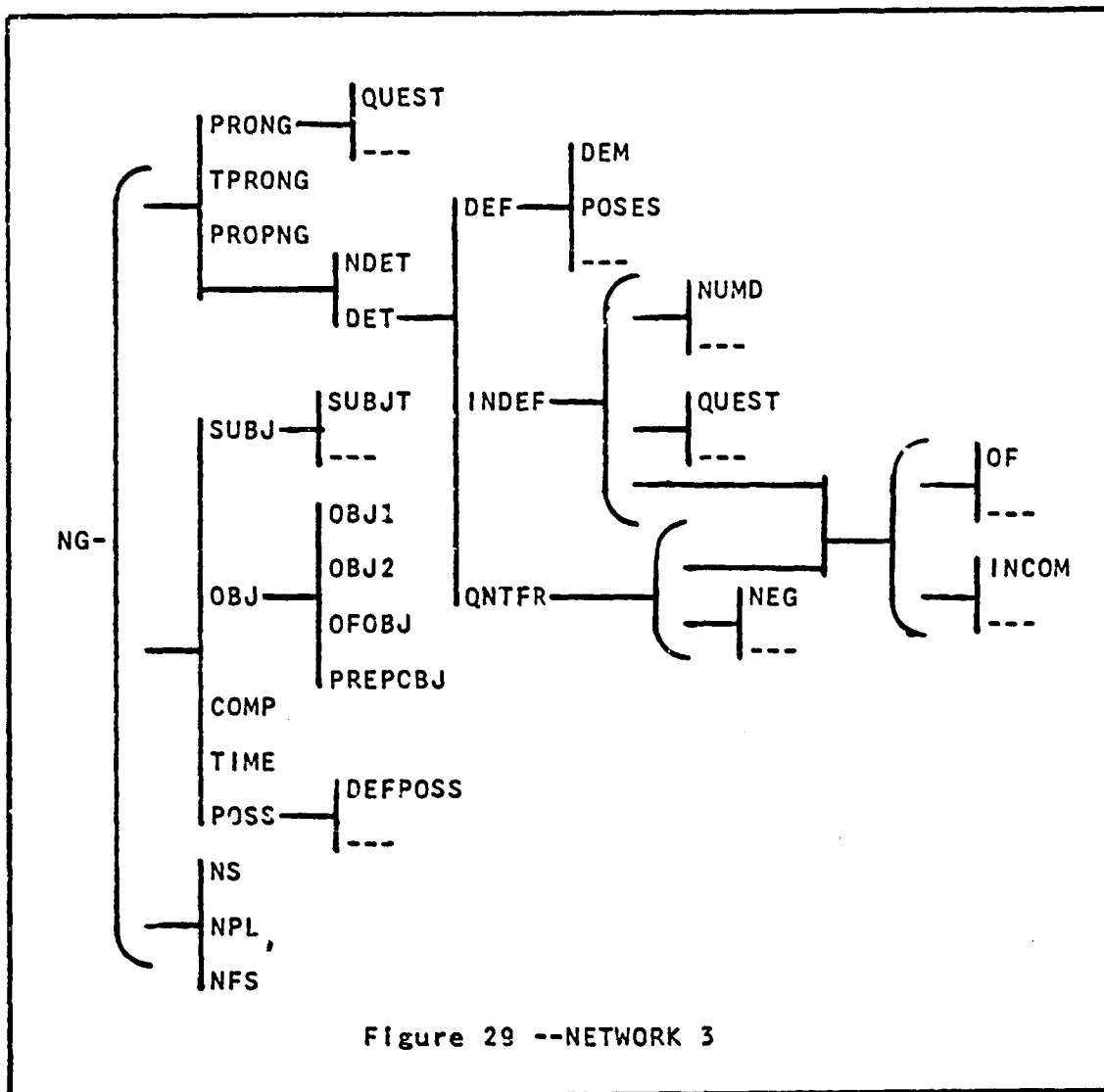
(s95) the next three days

Finally there is a NUMBER. It can either be a simple integer like "one", "two", etc. or a more complex construction such as "at least three", or "more than a thousand". It is possible for a NG to have all of its slots filled, as in:

DET	ORD	NUM	ADJ	ADJ	CLASF	CLASF	NOUN
the	first	three	old	red	city	fire	hydrants

Q(PREPG)	Q(CLAUSE)
without covers	you can find

It is also possible to have combinations of almost any subset. With these basic components in mind, let us look at the system network for NG in Figure 29.



First we can look at the major types of NG. A NG made up of a pronoun is called a PRONG. It can be either a QUESTION, like "who" or "what", or a non-question (the unmarked case) like "I", "them", "it", etc. The feature TPRONG marks a NG whose head is a special TPRON, like "something", "everything", "anything", etc. These enter into a peculiar construction

containing only the head and qualifiers, and in which an adjective can follow the head, as in:

(s95) anything green which is bigger than the moon

The feature PROPNG marks an NG made up of proper nouns, such as "John", "Oklahoma", or "The Union Of Soviet Socialist Republics."

These three special classes of NG do not have the structure described above. The PRONG is a single PRONoun, the PROPNG is a string of PROPNS, and the TPRONG has its own special syntax. The rest of the NGs are the unmarked (normal) type. They could be classified according to exactly which constituents are present, but in doing so we must be aware of our basic goals in systemic grammar. We could note whether or not a NG contained a CLASF or not, but this would be of minor significance. On the other hand, we do note, for example, whether it has a DET, and what type of DET it has, since this is of key importance in the meaning of the NG and the way it relates to other units. We distinguish between those with a determiner (marked DET) and those without one (NDET), as in:

(s97)	<u>Cats</u> adore <u>fish</u> .	NDET
(s98)	<u>The cat</u> adored <u>a fish</u> .	DET

The DET can be DEFINite (like "the" or "that"), INDEFInite (like "a" or "an"), or a quantifier (QNTFR) (like "some", "every", or "no"). The DEFINite determiners can be either DEMonstrative ("this", "that", etc.) or the word "the" (the unmarked case), or a POSSessive NG. The NG "the farmer's son"

has the NG "the farmer" as its determiner, and has the feature POSES to indicate this.

An INDEF NG can have a number as a determiner, such as:

- (s99) five gold rings
- (s100) at least a dozen eggs

In which case it has the feature NUMDET, or it can use an INDEF determiner, such as "a". In either case it has the choice of being a QUESTION. The question form of a NUMDET is "how many", while for other cases it is "which" or "what".

Finally, an NG can be determined by a quantifier (QNTFR). Although quantifiers could be subclassified along various lines, we do so in the semantics rather than the syntax. The only classifications used syntactically are between singular and plural (see below), and between NEGative and non-negative.

If a NG is either NUMD or QNTFR, it can be of a special type marked OF, like:

- (s101) three of the offices
- (s102) all of your dreams

An OF NG has a DETerminer, followed by "of", followed by a DEFinite NG.

A determined NG can also choose to be INCOMplete, leaving out the NOUN, as an

- (s103) Give me three.
- (s104) I want none.

Notice that there is a correspondence between the cases which can take the feature OF, and those which can be INCOM. We cannot say either "the of them" or "Give me the.". Possessives

are an exception (we can say "Give me Juan's." but not "Juan's of them"), and are handled separately (see below).

The middle part of Network 3 describes the different possible functions a NG can serve. In describing the CLAUSE, we described the use of an NG as a SUBJ, COMP, and OBJECTS of various types. In addition, it can serve as the object of a PREPG (PREPOBJ), in:

(s105) the rape of the lock

If it is the object of "of" in one of our special OF NGs, it is called an OFOBJ:

(s106) none of your tricks

A NG can also be used to indicate TIME, as in:

(s107) Yesterday the world ended.

(s108) The day she left, all work stopped.

Finally, a NG can be the POSSESSIVE determiner for another NG. In:

(s109) the cook's kettles

the NG "the cook" has the feature POSS, indicating that it is the determiner for the NG "the cook's kettle", which has the feature POSES.

When a PRONG is used as a POSS, it must use a special possessive pronoun, like "my", "your", etc. We can use a POSS in an incomplete NG, like

(s110) Show me yours.

(s111) John's is covered with mud.

There is a special class of pronouns used in these NG's

(labelled DEFPOSS), such as "yours", "mine", etc.

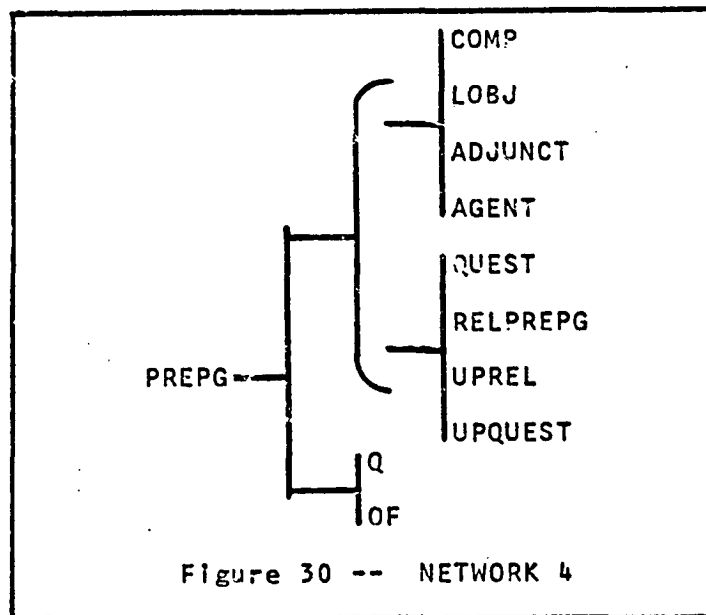
Continuing to the last part of Network 3, we see features of person and number. These are used to match the noun to the verb (if the NG is the subject) and the determiner, to avoid ungrammatical combinations like "these kangaroo" or "the women wins". In the case of a PRONG, there are special pronouns for first, second, and third person, singular and plural. The feature NFS occurs only with the first-person singular pronouns ("I", "me", "my", "mine"), and no distinction is made between other persons, since they have no effect on the parsing. All singular pronouns or other singular NGs are marked with the feature NS. The pronoun "you" is always treated as if it were plural and no distinction is made between "we", "you", "they", or any plural (NPL) NG as far as the grammar is concerned. Of course there is a semantic difference, which will be considered in later chapters.

2.3.6 Preposition Groups

The PREPG is a comparatively simple structure used to express a relationship. It consists of a PREPosition followed by an object (PREPOBJ), which is either a NG or a RSNG CLAUSE. In some cases, the preposition consists of a two or three word combination instead of a single word, as in:

- (s112) next to the table
 (s113) on top of the house

The grammar includes provision for this, and the dictionary lists the possible combinations and their meanings. The words in such a combination are marked as PREP2. The network for the PREPG is in Figure 30.



The PREPG can serve as a constituent of a CLAUSE in several ways. It can be a COMPLEMENT:

- (s114) Is it in the kitchen?

a locational object (LOBJ):

(s115) Put it on the table.

an ADJUNCT:

(s116) He got it by selling his soul.

or an AGENT:

(s117) It was bought by the devil.

If the PREPG is a constituent of a QUESTION CLAUSE, it can be the question element by having a QUEST NG as its object:

(s118) In what city

(s119) for how many days

(s120) by whom

In which case the PREPG is also marked QUEST. A PREPREL CLAUSE contains a RELPREPG:

(s121) the place in which she works

If the CLAUSE is an UPQUEST or an UPREL, the PREPG can be the constituent which is "missing" the piece which provides the upward reference. In this case it is also marked UPREL:

(s122) the lady I saw you with

or UPQUEST:

(s123) Who did you knit it for?

In these cases, it is also marked SHORT to indicate that the object is not explicitly in the PREPG. It can also be short if it is a PREPG in a DANGLING PREPG or PREPREL CLAUSE:

(s124) what do you keep it in?

Within a NG, a PREPG serves as a qualifier (Q):

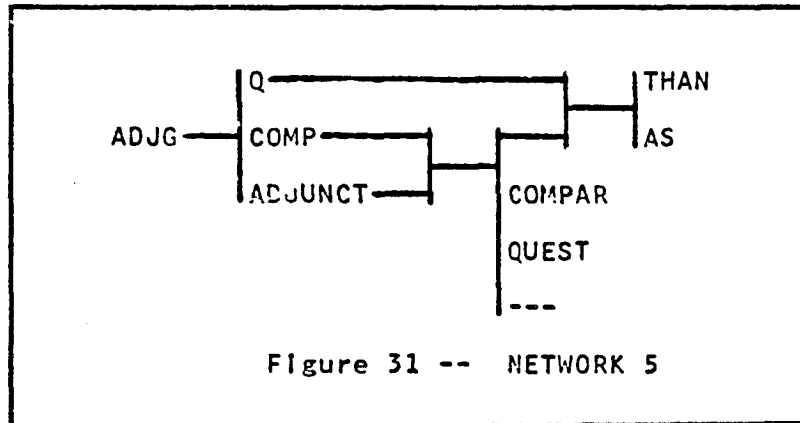
(s125) the man in the iron mask

or as the body of an OF NG:

(s126) some of the people

2.3.7 Adjective Groups

The ADJG is a specialized unit serving as a COMPLEMENT of an intensive clause, as a Qualifier to an NG, or as a CLAUSE ADJUNCT. The network is:



An ADJG which serves as an ADJUNCT contains an adverb, like "fast" in:

(s127) He could run faster than an arrow.

In place of an adjective. (Clearly our terminology could do with some cleaning up at places like this in doing a theoretical version of the grammar.) The other two types of ADJG use an adjective, as in a Qualifier:

(s128) a hotel as bad as the other one

or a COMPLEMENT:

(s129) They were blissful.

The basic forms for an ADJG include THAN:

(s130) holier than thou

AS:

(s131) as quick as a flash

COMPARative:

(s132) This one is bigger.

or QUESTIon:

(s133) How well can he take dictation?

The network is arranged to show that a qualifier ADJG can be only of the first two forms -- we cannot say "a man bigger" without using "than", or say "a man big". In the special case of a TPRON such as "anything" as in:

(s134) anything strange

the word "strange" is considered an ADJ which is a direct constituent of the NG, rather than an ADJG.

The grammar does not yet account for more complex uses of the word "than".

2.3.8 Verb Groups

The English verb group is designed to convey a complex combination of tenses so that an event can relate several time references. For example, we might have:

(s135) By next week you will have been living
here for a month.

This is said to have the tense "present in past in future". Its basic reference is to the future -- "next week", but it refers back to the past from that time, and also indicates that the event is still going on. This type of recursive tense structure has been analyzed by Halliday <Halliday 1966b> and our grammar adopts a variant of his scheme.

Essentially the choice is between four tenses, PAST, PRESENT, FUTURE, and MODAL. Once a choice between these has been made, a second, third, fourth, and even fifth choice can be made recursively. The combination of tenses is realized in the syntax by a sequence of the auxiliary verbs "be", "have", and "going to", along with the ING, EN, and INFINITIVE forms of the verbs. The restrictions on the recursion are:

1. PRESENT can occur only at the outer ends of the series (at first and/or final choice).
2. Except in the final two positions, the same tense cannot be selected twice consecutively.
3. Future can occur only once other than in last position.
4. Modal can be only in final position.

It is important to distinguish between the position of a word in the VG and the position of its tense in the recursive tense feature -- the direction is reversed. In s135, "will" is the first word, and "living" the last, while the tense is PRESENT in PAST in FUTURE. Some sample verb groups and their tenses are: (from <Halliday 1966b>)

ACTIVE	
took	- past
takes	- present
will take	- future
can take	- modal
has taken	- past in present
was taking	- present in past
was going to have taken	- past in future in past
was going to have been taking	- present in past in future in past
PASSIVE	
is taken	- present
could have been taken	- past in modal
has been going to have been taken	-
	past in future in past in present

Figure 32 -- Verb Group Tenses

The structure of a finite VG (one taking part in this tense system -- see below for other types) is a sequence of verbs and auxiliaries in which the last is the "main verb" (marked MVB and remembered by the parser), and the first is either a MODAL, the word "will", or a "finite" verb (one carrying tense and number agreement with the subject). Interspersed in the sequence there may be adverbs, or the word "not" (or its reduced form "n't"). The best way to describe the relationship between

the sequence of verbs and the tense is by giving a flow chart for parsing a VG. This is a good example of the usefulness of representing syntax in the form of procedures, as it describes a relatively complex system in a clear and succinct way.

In the flow chart (Figure 33) the variable T represents the tense, and the symbol "." indicates the addition of a member to the front of a list. The "=" indicates replacement in the FORTRAN sense, and the function "REMOVE" removes words from the input string. The features used are those described for verbs in section 2.3.9. The command (FQ PASV) indicates that the entire VG is to be marked with the feature PASV (passive voice). The flow chart does not indicate the entire parsing, but only that part relevant to determining the tense.

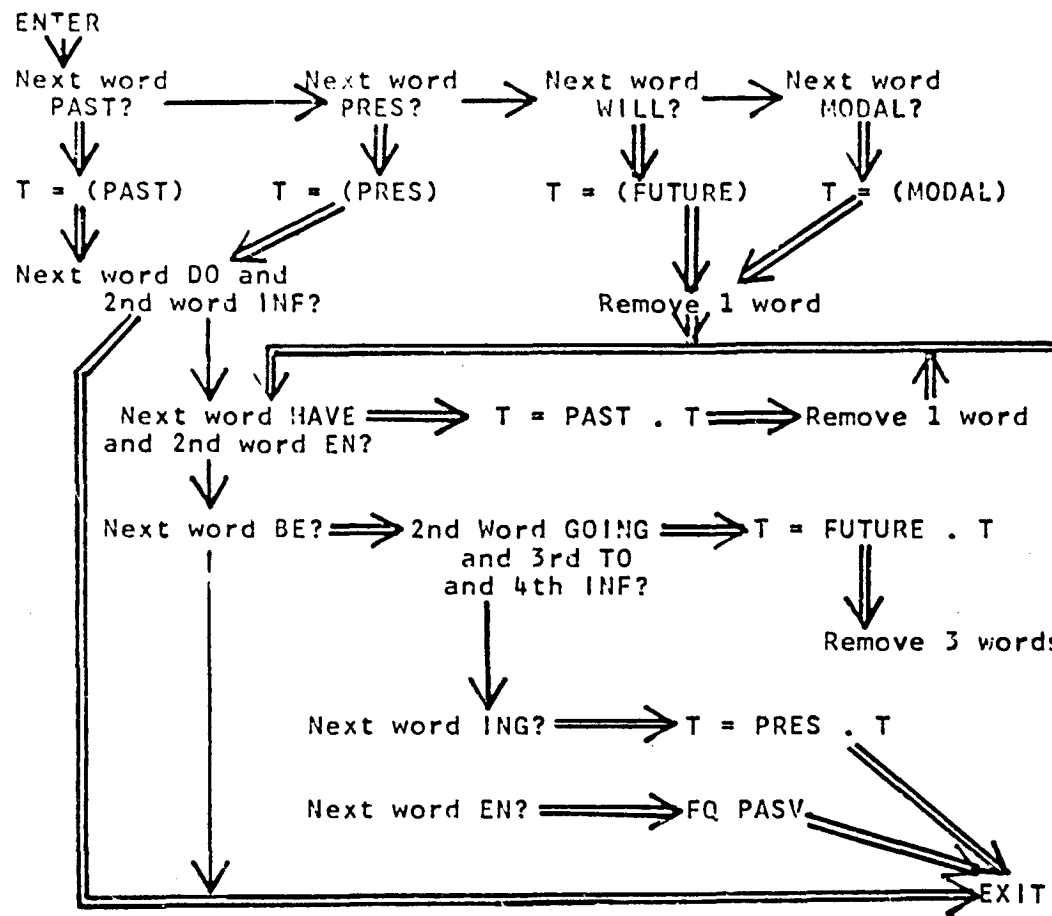
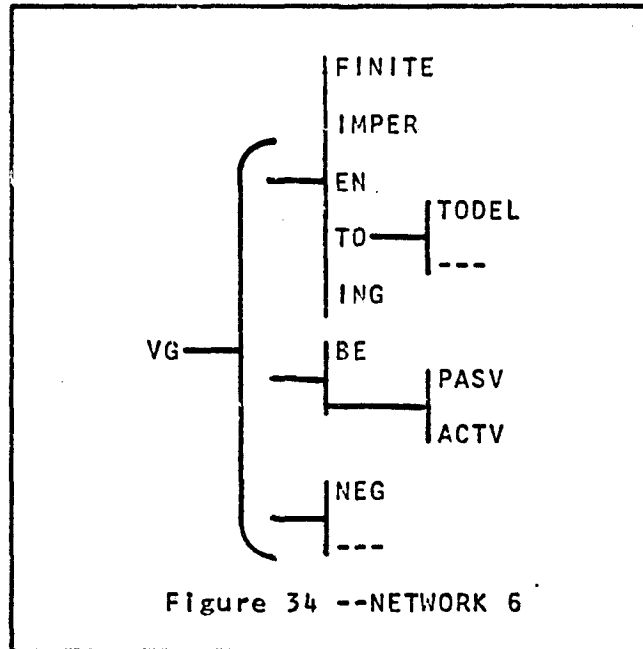


Figure 33 -- Syntax of VG Tense Structure

This system of tenses is operative only for FINITE verb groups. The network for the VG in general is:



There are several types of VG which do not enter the normal tense system, but which have a specialized form. The IMPER VG is used in imperatives:

(s136) Fire when ready.

(s137) Don't drop the baby.

It consists of a verb in the INFINITIVE form, possibly preceded by the auxiliary "do" or its negative form "don't". The EN VG is used in EN RSQ CLAUSES, like:

(s138) a man forsaken by his friends

and consists of a past participle verb. The ING VG is made up of an ING verb or the verb "being" followed by an EN verb. It is

used in various types of ING clauses:

(s139) Being married is great.

(s140) the girl sitting near the wall

Similarly, the TO VG is used in TO clauses. In the case of conjoined structures, the "to" may be omitted from the second clause, as in:

(s141) We wanted to stop the war and end repression.

Such a VG is marked TODEL.

We separate those verb groups whose main verb is "be" from the others, as they do not undergo the further choice between PASV and ACTV. These correspond to the same features for clauses, and are seen in the structure by the fact that a PASV VG contains a form of the auxiliary "be" followed by the main verb in the EN form, as in:

(s142) The paper was finished by the deadline.

(s143) He wanted to be kissed by the bride.

Finally, any VG can be NEGative, either by using a negative form of an auxiliary like "don't", "hasn't", or "won't", or by including the word "not".

2.3.9 Words

Our grammar uses a number of separate word classes, each of which can be divided into subclasses by the features assigned to individual words. It was necessary to make arbitrary decisions as to whether a distinction between groups of words should be represented by different classes or different features within the same class. Actually we could have a much more tree-like structure of word classes, in which the ideas of classes and features were combined. Since this has not yet been done, we will present a list of the different classes in alphabetical order, and for each of them give descriptions of the relevant features. Many words can be used in more than one class, and some classes overlap to a large degree (such as NOUN and CLASF). In our dictionary, we simply list all of the syntactic features the word has for all of the classes to which it can belong. When the parser parses a word as a member of a certain class, it sorts out those features which are applicable. Figure 35 is a list of the word classes and their features.

ADJ -- Adjective is one of the constituents of a NG as well as being the main part of an ADJG. This class includes words like "big", "ready", and "strange". The only features are SUPERlative (as in "biggest") and COMPARative (as in "bigger").

ADV -- We use the name "adverb" to refer to a whole group of words used to modify other words or clauses. It is sort of a

CLASS	FEATURES
ADV	ADV ADVADV LOBJ PLACE PREPADV TIMW TIM2 VBAD
BINDER	BINDER
CLASF	CLASF
DET	DEF DEM DET INCOM INDEF NEG NONUM NPL NS OFD PART QDET QNTR
ADJ	ADJ COMPAR SUP
NOUN	MASS NOUN NPL NS POSS TIME TIM1
NUM	NPL NS NUM
NUMD	NUMD NUMDALONE NUMDAN NUMDAT
ORD	ORD TIMORD
PREP	PLACE PREP NEED2
PREP2	PREP2
PRON	DEFPOSS NEG NFS NPL NS OBJ POSS PRON REL SUBJ
PRONREL	NPL NS PRONREL
PROPN	NPL NS POSS PROPON
PRT	PRT
QADJ	PLACE QADJ
TPRON	NEG NPL NS TPRON
VB	AUX BE DO EN HAVE IMPERF INF ING INGOB INGOB2 INT ITRNS ITRNSL MODAL MVB NEG PAST PRES QUAX REPOB REPOB2 SUBTOB SUBTOB2 TOOB TOOB2 TO2 TRANS TRANSL TRANSL2 TRANS2 VB VFS VPL VPRT V3PS WILL

Figure 35 -- Word Classes and Applicable Features

"mixed bag" of words which don't really fit anywhere else.

The basic classification depends on what is being modified, and has the terms (ADVADV VBAD PREPADV CLAUSEADV). An ADVADV is a word like "very" which modifies other adverbs and adjectives. A VBAD modifies verbs, and includes the class of words ending in "-ly" like "quickly" and "easily". A PREPADV modifies prepositions, as "directly" in "directly above the stove". A CLAUSEADV is a constituent of a clause, and can be either TIMW or PLACE. A TIMW like "usually", "never", "then", or "often" appears as a CLAUSE constituent specifying the time. The PLACE ADV "there" can either be an adjunct, as in:

(s144) There I saw a miracle.

or an LOBJ, as in:

(s145) Put it there.

BINDER -- Binders are used to "bind" a secondary clause to a major clause, as in:

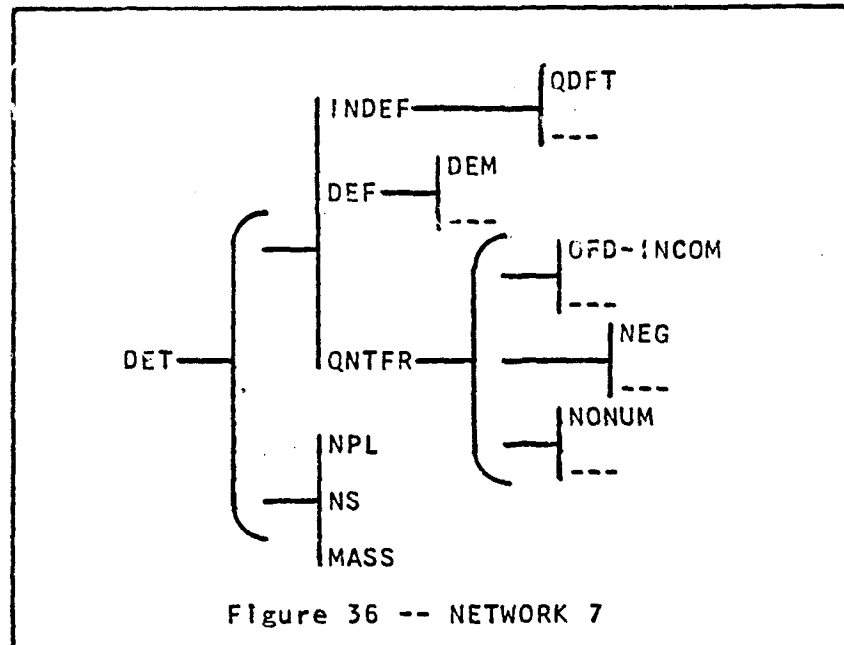
(s146) Before you got there, we left.

(s147) I'll go if you do.

We do not assign any other features to binders.

CLASF -- In Section 2.3.5 we discussed the use of CLASF as a constituent of a NG. The CLASF is often another NOUN, but it appears in a position like an adjective, as in "boy scout".

DET -- DETerminers are used as a constituent of a NG, as described in 2.3.5. They can have a number of different features, as described in the network:



A DET can be INDEFinite, like "a" or "an" or the question determiners (QDET) "which", "what", and "how many". It can be DEFinite, like "the" or the DEMonstrative determiners "this", "that", "those", and "these". Or it can be a quantifier (QNTFR) like "any", "every", "some", etc. Quantifiers can have the feature OFD, indicating that they can be used in an OF NG like:

(s148) some of my best friends

We originally had a separate feature named INCOM indicating whether they could be used in an Incomplete NG like:

(s149) Buy some.

but later analysis showed these features were the same. Not all quantifiers are OFD -- we cannot say "every of the cats" or "Buy every." Quantifiers can also be NEGative, like "none"

or "no", or can be NONUM, indicating that they cannot be used with a number, such as "many" or "none" (we can say "any three cats" or "no three cats", but not "none three" or "many three"). The NG program takes these features into account in deciding what NG constituents to look for. It also has to find agreement in number between the DET and the NOUN. A DET can have the features "singular" (NS), "plural" (NPL) or MASS (like "some" or "no", which can go with MASS nouns like "water"). A DET can have more than one of these -- "the" has all three, while "all" is MASS and NPL, and "a" is just NS.

NOUN -- The main constituent of a NG is its NOUN. It has a feature of number, identical to that of the DETerminers it must match. The word "parsnip" is NS, "parsnips" is NPL, and "wheat" is MASS. Some nouns may have more than one of these, such as "fish", which is all three since it can be used in "a fish", "three fish", or "Fish is my favorite food." In addition, a NOUN can be POSSessive, like "parsnip's".

In order to tell whether a NG is functioning as a time element in a CLAUSE, we need to know whether its NOUN can refer to time. We therefore have two features -- TIME words like "day", and "month", as in:

(s150) The next day it started to snow.
and TIM1 words like "yesterday" and "tomorrow". This illustrates the interaction between syntax and semantics. A

phrase like "the next visit" can be used to indicate a time, since a "visit" is an event. The actual distinction should be the semantic difference between "event" and "non-event"

The grammar could be easily changed to look at the semantic features rather than syntactic features of the NOUN in deciding whether it could be the head of a TIME NG.

NUM -- The class of NUMbers is large (uncountably infinite) but not very interesting syntactically. For our purposes we only note the features NS (for "one") and NPL (for all the rest). In fact, our system does not accept numbers in numeric form, and has only been taught to count to ten.

NUMD -- In complex number specifications, like "at least three" or "more than a million", there is a NUMD. The features they can have are (NUMDAN NUMDAS NUMDAT NUMDALONE). NUMDAN words such as "more" and "fewer" are used with "than", while NUMDAS words such as "few" fit into the frame "as...as", and NUMDATs are preceded by "at", as in "at least", and "at most". NUMDALONE indicates that the NUMD can stand alone with the number, and includes "exactly" and "approximately".

ORD -- The class of ORDinals includes the ordinal numbers "first", "second", etc., and a few other words which can fit into the position between a determiner and a number, like "next", "last", and "only". Notice that SUPERlative ADJectives can also fill this slot in the NG.

PREP -- Every PREPG begins with a PREPosition, either alone, or

as part of a combination such as "on top of". In the combination case, the words following the initial PREP have the feature PREP2. A PREP which cannot appear without a PREP2 (such as "next" which appears in "next to") are marked NEED2.

PRON -- PRONouns can be classified along a number of dimensions, and we can think of a large multi-dimensional table with most of its positions filled. They have number features (NS NPL NFS) (note that instead of the more usual division into first, second, and third person, singular and plural, we have used a reduced one in which classes with the same syntactic behavior are lumped together). They can be POSSessive, such as "your" or "my", or POSSDEF, like "yours" or "mine". Some of the personal pronouns distinguish between a SUBject form like "I" and an OBJect form like "me". there are also special classes like DEMonstrative ("this" and "that") and PRONREL -- the pronouns used in relative clauses, such as "who", "which", and "that". Those which can be used as a question element, such as "which" and "who" are marked QUEST.

PROPN -- Proper nouns include single words like "Carol", or phrases such as "The American Legion" which could be parsed, but are interpreted as representing a particular object (physical or abstract). A PROPN can be NPL or NS, and is assumed to be NS unless defined otherwise.

PRT -- In Section 2.3.4, we discussed clauses which use a

combination of a "particle" and a verb, like "pick up" or "knock out". The second word of these is a PRT.

QADJ -- One class of QUESTION CLAUSE uses a QADJ such as "where", "when", or "how" as its question element. They can also be used in various kinds of relative clauses, as explained in Section 2.3.3.

TPRON -- There is a small class of words made up of a quantifier and the suffix "-thing" which enter into a special type of NG construction like "anything green". This is not an abbreviation for a quantifier followed by a noun, since the NG "any block green" would have the same structure but is not grammatical.

VB -- The verb has the most complex network of features of any word in our grammar. They describe its tense, transitivity, number, and use, as well as marking special verbs like "be". The network is in Figure 37.

Verbs are divided into AUXiliaries and others (unmarked). AUXiliaries are the "helping verbs" which combine with others in complex VG structures. They can have special NEGative forms, like "can't", or can appear standing alone at the beginning of a QUESTION, in which case they have the function QAUX, as in:

(s151) Will I ever finish?

The auxiliaries include "be", "do", "have", "will", and the MODALS like "could", "can", and "must". Separate features are

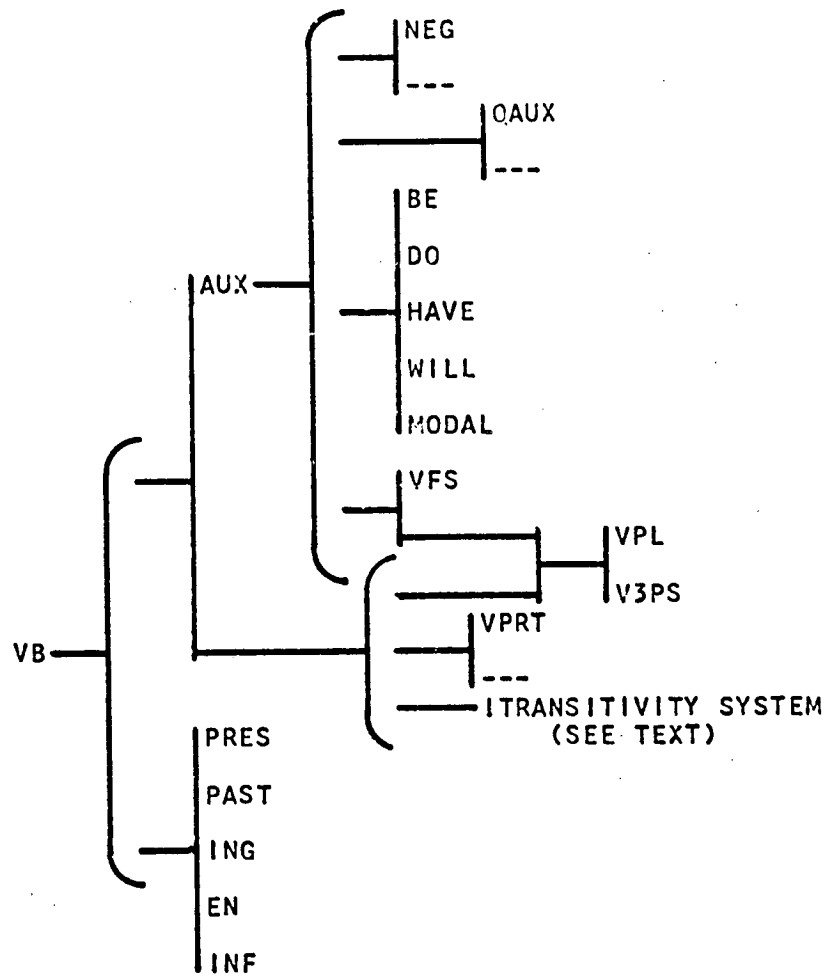


Figure 37 -- NETWORK 8

used for these as they are critical in determining the structure of a VG. An AUX can choose from the system of person and number, distinguishing "third-person singular" (V3PS) as in "is", "plural", as in "have", or "first singular" (VFS), used only for "am".

Non-auxiliary verbs can be VPRT, which combine with a PRT, and they have a whole cluster of transitivity features. In Section 2.3.4 we described the different transitivity features of the CLAUSE, and these are controlled by the verb. We therefore have the features (TRANS ITRNS TRANS2 TRANS1 ITRNSL INT) In addition, the verb can control what types of RSNGL CLAUSE can serve as its various objects. The feature names combine the type of CLAUSE (ING TO REPORT SUBTO SUPING) with either -OB or -OB2, to get a product set of features like SUBTOB and INGOB2.

For example, the verb "want" has the features T00B and SUBTOB, but not INGOB, REPOB, etc. since "I want to go." and "I want you to go." are grammatical, but "I want going.", "I want that you go.", etc. are not.

Finally, all of these kinds of verbs can be in various forms such as ING ("breaking"), EN ("broken"), INFINITIVE ("break), PAST ("broke"), and PRESENT ("breaks"). The network does not illustrate all of the relations, as some types (like MODAL) do not make all of these choices.

2.3.10 Conjunction

One of the most complex parts of English is the system of conjunction. This section presents a simplified version which has been implemented using the special interrupt feature of PROGRAMMAR (see Section 2.4.2 for details). This makes the parsing particularly simple.

The basic concept is that any unit in a sentence can be replaced by a COMPOUND unit of the same type. In the sentence:

(s152) I baked a chocolate cake, three pies, and some hashish brownies.

the object is a COMPOUND NG with three components. There can be a compound ADJ, as in:

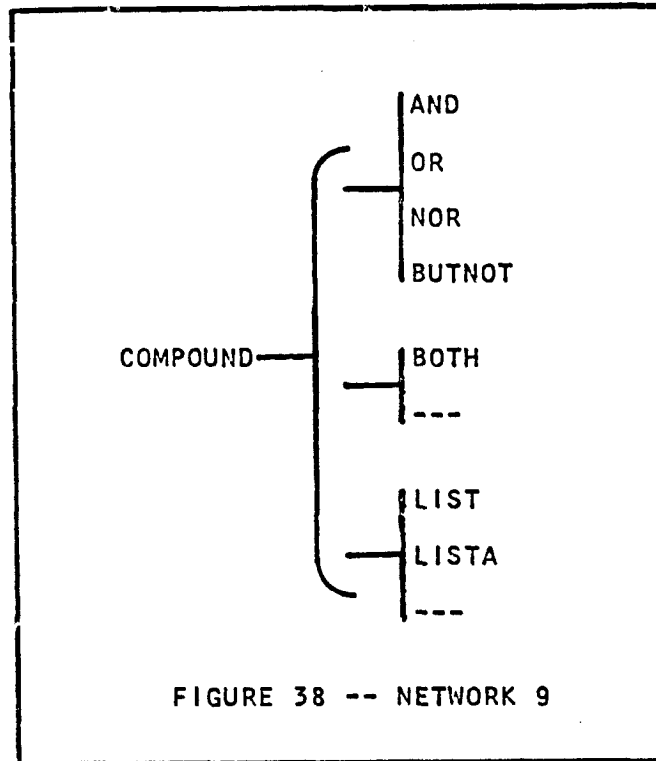
(s153) a red or yellow flag

or a phrase can be ambiguous, as in:

(s154) black cats and horses

This can be interpreted as having either a COMPOUND NG, composed of the NGs "black cats" and "horses", or a single NG with a COMPOUND NOUN, "cats and horses".

The features of a COMPOUND unit are determined by its components and by the type of conjunction. The conjunction features are from the following network:



The first choice is the actual conjunction used. The feature BOTH indicates a word at the beginning of a COMPOUND structure, as in:

(s155) both you and your family

the specific word depends on the conjunction -- "both" with "and", "either" with "or", and "neither" with "nor". The features BOTH and NOR combine in:

(s156) neither you nor I

A conjoined structure can be made up of two elements with a conjunction (as in the previous three examples), or a LIST connected with commas and a conjunction before the last element (as in s152), or it can be a list connected with conjunctions (a

LISTA), as in:

(s157) cabbages and kings and sealing wax and things

Every constituent but the first is marked with the feature COMPONENT. The COMPOUND unit also takes on features from its constituents. It may have features such as number and tense, relevant to its syntactic function. For example, a COMPOUND NG with the feature AND must be plural (NPL), while one with the feature OR will have the number feature of its initial constituent.

Finally, there can be "forked" structures like:

(s158) We dropped the watermelon and ran.

Notice that this cannot be viewed in the same way, since "dropped the watermelon" is not a single constituent of any unit. This sentence is viewed as a conjunction of two clauses, sharing the subject "we". The second clause is marked with the feature SUBJFORK to indicate this. Similarly, the subject and verb can be shared, as in:

(s159) He gave Greg the apple and Gloria the pear.

The clause "Gloria the pear" is marked as VBFORK and SUBJFORK. The only constituents relative to a fork are those necessary for the clause structure. in:

(s160) They easily saw the trouble and cheerfully fixed it.

the parser does not determine whether the "easily" applies to both constituents. but leaves this decision to semantics. This is in keeping with our orientation of syntax to surface

structure and the belief that much of "deep structure" is more the province of the semantic analyzer.

2.3.11 Following the Parser in Operation

Let us follow the parser through two examples to see how the grammar is used in practice. We will not actually watch all of the details, or deal with the way semantic programs are intermixed with the grammar. Instead we will follow a somewhat reduced version, to get a feeling for the way the grammar works, and the way it interacts with the different features described above. We have chosen one very simple sentence, and another which is difficult enough to exercise some of the more complex features of the grammar. The first sentence is the first sentence of our sample dialog (Section 1.3):

Pick up a big red block.

The system begins trying to parse a sentence, which, as explained above, means looking for a MAJOR CLAUSE. It activates the grammar by calling (PARSE CLAUSE MAJOR). Since CLAUSE is one of our units, there is a program defined for it. The CLAUSE program is called with an initial feature list of (CLAUSE MAJOR).

The CLAUSE program looks at the first word, in order to decide what unit the CLAUSE begins with. If it sees an adverb, it assumes the sentence begins with a single-word modifier. If it sees a PREPosition, it looks for an initial PREPG. If it sees a BINDER, it calls the CLAUSE program to look for a BOUND CLAUSE. In English (and possibly all languages) the first word of a construction often gives a very good clue as to what that

construction will be. We have "advance notice" of what structures to look for, and this makes parsing much easier. Our grammar tries to make as much use as possible of these natural "signals". In this case, the initial word is a verb, and this indicates that we may have an IMPERATIVE CLAUSE. The program calls (PARSE VG IMPER) to start the VG program with the initial VG feature list (VG IMPER), looking for a VG of the right type. The VG program checks this initial feature list, and sees that it is looking for an IMPERative VG. This must either begin with some form of the verb "do", or with the main verb itself. Since the next word is not "do", it calls (PARSE VB INF (MVB)). This is a different kind of call to PARSE, since VB is not a unit we have defined. It is a word class, and the call says to check the next word in the input (In this case still the first word) to see whether it is indeed the INFinitive form of a VerB. If so, it is to be attached to the parsing tree, and given the additional feature MVB (main verb). The current structure can be diagrammed:

(CLAUSE MAJOR)	(pick...)
(VG IMPER)*	(pick...)
(VB MVB INF TRANS VPRT)	pick

Figure 39 -- Syntactic Structure 1

We use several conventions for diagramming syntactic structures. Rather than using a tree format (which quickly grows off of the page), we use a format more like a traditional outline, with the sub-phrases of any phrase indicated

Immediately below it and indented. We use the symbol "*" to indicate the program which is currently active, and show the actual words of each piece of the sentence to the right of the outline, putting units larger than WORD in parentheses. A series of dots in the English words to the right of a unit indicates that the program for that unit has not yet finished.

Figure 59 shows that we have a CLAUSE, with a constituent which is a VG, and that the VG program is active. The VG so far consists of only a VB. Notice that some new properties have appeared on the list for VB. We have not mentioned TRANS or VPRT. These came from the definition of the word "pick" when we called the function PARSE for a word (see section 2.4.4 for details).

Ordinarily, the VG program checks for various kinds of tense and number, but in the special case of an IMPER VB, it returns immediately after finding the verb. We will see other cases in the next example.

When the VG program succeeds, CLAUSE takes over again. Since it has found the right kind of VG for an IMPERative CLAUSE, it puts the feature IMPER on the CLAUSE feature list. It then checks to see whether the MVB has the feature VPRT, indicating it is a special kind of verb which takes a particle. It discovers that "pick" is such a verb, and next checks to see if the next word is a PRT, which it is. It then checks in the dictionary to see if the combination "pick up" is defined, and

when it discovers this is true, it calls (PARSE PRT) to add "up" to the parsing tree. Notice that we might have let the VG program do the work of looking for a PRT, but it would have run into difficulties with sentences like "Pick the red block up." in which the PRT is displaced. By letting the CLAUSE program do the looking, the problem is simplified.

As soon as it has parsed the PRT, the CLAUSE program marks the feature PRT on its own feature list. It then looks at the dictionary entry for "pick up" to see what transitivity features are there. It is TRANSitive, which indicates that we should look for one object -- OBJ1. The dictionary entry does not indicate that this is a verb which can take special types of RSNG clauses as objects, so the object must be either a NG or a WHRS clause (which can appear wherever a NG can). If the object were a WHRS clause, it would begin with a relative pronoun, like "Pick up what I told you to." Since the next word is "a", this is not the case, so the CLAUSE program looks for an object by calling (PARSE NG OBJ OBJ1), asking the NG program to find a NG which can serve as an OBJ1. The structure is now:

(CLAUSE MAJOR IMPER PRT)	(pick up...)
(VG IMPER)	(pick)
(VB MVB INF TRANS PRT)	pick
(PRT)	up
(NG OBJ OBJ1)*	(...)

Figure 40 -- Syntactic Structure 2

The NG program is started and notices that the upcoming

word is a DET, "a". It calls (PARSE DET) to add it to the parsing tree, then uses the function TRNSF to transfer relevant features from the DET to the entire NG. It is interested in the type of determination (DEF vs. INDEF vs. QNTR), and the number (NS vs. NPL). It also adds the feature DET to the NG to indicate that it has a determiner. The feature list for the NG is now:

(NG OBJ OBJ1 DET INDEF NS)

since "a" is a singular indefinite determiner. The NG program then notices the feature INDEF, and decides not to look for a number or an ordinal (we can't say "a next three blocks"), or for the OF construction ("a of them" is impossible). It goes on immediately to look for an ADJective by calling (PARSE ADJ). When this succeeds with the next word "big", a simple program loop returns to the (PARSE ADJ) statement, which succeeds again with "red". on the next trip it fails, and sends the program on to look for a classifier, since "block" isn't an ADJ. But "block" isn't a CLASF either in our dictionary, so the NG program goes on to look for a NOUN, by calling (PARSE NOUN). This succeeds with the NOUN "block", which is singular, and the program checks to see if it agrees with the number features already present from the determiner (to eliminate illegal combinations like "these boy"). In this case, both are singular (NS), so the program is satisfied. Ordinarily it would go on to look for qualifiers, but in this case there is nothing left in

the sentence. Remember that we have an especially easy way of indicating in a PROGRAMMAR program what action should be taken at any point if the sentence runs out. We can do it by simply putting a third direction in any branch statement. In this case, since we have found all of the basic constituents we need for a NG, the "third branch" tells us that the NG program should return success. If we had run out after the determiner, it would have sent us to check for an INCOMPLETE NG, while if we had run out after an ADJ it would have entered a backup program which would check to see whether it had misinterpreted a NOUN as an ADJ.

In this case, the NG program returns, and the CLAUSE program similarly notices that the sentence has ended. Since a TRANS verb needs only one object, and that object has been found, the CLAUSE program marks the feature TRANS, and returns, ending the parsing. In actual use, a semantic program would be called here to understand and execute the command -- in fact, semantic programs would have been called at various points throughout the process. The final result looks like:

(CLAUSE MAJOR IMPER PRT TRANS) (pick up a big red block)

(VG IMPER)	(pick)
(VB MVB INF TRANS VPRT)	pick

(PRT)	up
-------	----

(NG OBJ OBJ1 DET INDEF NS)	(a big red block)
(DET INDEF NS)	a
(ADJ)	big
(ADJ)	red
(NOUN NS)	block.

Figure 41 -- Syntactic Structure 3

Now let us take a more complex sentence, like:

How many blocks are supported by the cube which I
wanted you to pick up?

We will not go into as much detail, but will emphasize the new features exhibited by this example. First, the parser recognizes that this sentence is a question by its punctuation. It ends with a question mark. This "cheating" is not really necessary, and in the future the grammar will be revised to look for the other signals of a question (for example, beginning with a determiner like "how many" or "which").

In any event, the feature QUESTION is noted, and the program must decide what type of question it is. It checks to see if the CLAUSE begins with a QADJ like "why", "where", etc. or with a PREPosition which might begin a PREPG QUEST (like "In what year...").

All of these things fail in our example, so it decides the CLAUSE must have a NG as its question element, (called NGQ), marks this feature, and calls (PARSE NG QUEST). The NG program starts out by noticing QUEST on its initial feature list, and looking for a question determiner (DET QDET). Since there are only three of these ("which", "what", and "how many"), the program checks for them explicitly, parsing "how" as a QDET, and then calling (PARSE NIL MANY), to add the word "many" to the parsing tree, without worrying about its features. (The call (PARSE NIL X) checks to see if the next word is actually the word "x")).

Since a determiner has been found, its properties are added to the NG feature list, (in this case, (NUMDET INDEF NPL)), and the NG program goes on with its normal business, looking for adjectives, classifiers, and a noun. It finds only the NOUN "blocks" with the features (NOUN NPL). The word "block" appears in the dictionary with the feature NS, but the input program which recognized the plural ending changed NS to NPL for the form "blocks". Agreement is checked between the NOUN and the rest of the NG, and since "how many" added the feature NPL, all is well. This time, there is more of the sentence left, so the NG program continues, looking for a qualifier. It checks to see if the next word is a PREposition (as in "blocks on the table), a relative word ("blocks which...), a past participle ("blocks supported by...), an ING verb ("blocks sitting on...) a comparative adjective ("blocks bigger than...) or the word "as" ("blocks as big as...). If any of these are true, it tries to parse the appropriate qualifying phrase. If not, it tries to find an RSQ CLAUSE ("blocks the block supports). In this case, all of these fail since the next word is "are", so the NG program decides it will find no qualifiers, and returns what it already has. This gives us:

(CLAUSE MAJOR QUESTION NGQ)*	(how many blocks...)
(NQ QUEST DET NUMDET NPL INDEF)	(how many blocks)
(DET QDET NPL INDEF)	how
()	many
(NOUN NPL)	blocks

Figure 42 -- Syntactic Structure 4

Next the CLAUSE program wants a VG, so it calls (PARSE VG NAUX). The feature NAUX indicates that we want a VG which does not consist of only an AUXiliary verb, like "be" or "have". If we saw such a VG, it would indicate a structure like "How many blocks are the boxes supporting?", in which the question NG is the object of the CLAUSE. We are interested in first checking for the case where the question NG is the subject of the CLAUSE.

The VG program is designed to deal with combinations of auxiliary verbs like "had been going to be..." and notes that the first verb is a form of "be". It calls (PARSE VB AUX BE), assuming that "are" is an auxiliary rather than the main verb of the sentence (if this turns out wrong, there is backup). It transfers the initial tense and person features from this verb to the entire VG (The English VG always uses the leading verb for these features, as in "He has been...", where it is "has" which agrees with "he") in this case "are" is plural (VPL) and present tense (PRES).

When "be" is used as an auxiliary, it is followed by a verb in either the ING or the EN form. Since "supported" is an EN form (and was marked that way by the Input program), The VG program calls (PARSE VB EN (MVB)), marking "supported" as the main verb of the clause. The use of a "be" followed by an EN form indicates a PASV VG, so the feature PASV is marked, and the VG program is ready to check agreement. Notice that so far we haven't found a SUBJECT for this clause, since the QUESTION NG

might have been an object, as in "How many blocks does the box support?" However the VG program is aware of this, and realizes that instead of checking agreement with the constituent marked SUBJ, it must use the one marked QUEST. It uses PROGRAMMAR's pointer-moving functions to find this constituent, and notes that it is NPL, which agrees with VPL. VG therefore is happy and returns its value. We now have:

(CLAUSE MAJOR QUESTION NGQ)*	(how many blocks are supported...)
(NG QUEST DET NUMDET NPL INDEF)	(how many blocks)
(DET QDET NPL INDEF)	how
()	many
(NOUN NPL)	blocks
(VG NAUX VPL PASV (PRES))	(are supported)
(VB AUX BE PRES VPL)	are
(VB MVB EN TRANS)	supported

Figure 43 -- Syntactic Structure 5

The CLAUSE program resumes, and marks the feature SUBJQ, since it found the right kind of VG to indicate that the NG "how many blocks" is indeed the subject. It next checks to see if we have a PRT situation as we did in our first example. We don't, so it next checks to see if the VG is PASV, and marks the clause with the feature PASV. This indicates that there will be no objects, but there might be an AGENT phrase. It checks that the next word is "by", and calls (PARSE PREPG AGENT).

The PREPG program is fairly simple -- it first calls (PARSE PREP), then (PARSE NG OBJ PREPOBJ). The word "by" is a PREP, so the first call succeeds and NG is called and operates as

described before, finding the DET "the" and the NOUN "cube", and checking the appropriate number features. In this case, "the" is both NPL and NS, while "cube" is only NS, so after checking the NG has only the feature NS.

The NG program next looks for qualifiers, as described above, and this time it succeeds. The word "which" signals the presence of a RSQ WHRS CLAUSE modifying "cube". The NG program therefore calls (PARSE CLAUSE RSQ WHRS). The parsing tree now looks like:

(CLAUSE MAJOR QUESTION NGQ SUBJQ PASV)	
	(how many blocks are supported by the cube...)
(NG QUEST DET NUMDET NPL INDEF)	(how many blocks)
(DET QDET NPL INDEF)	how
()	many
(NOUN NPL)	blocks
(VG NAUX VPL PASV (PRES))	(are supported)
(VB AUX BE PRES VPL)	are
(VB MVB EN TRANS)	supported
(PREPG AGENT)	(by the cube...)
(PREP)	by
(NG OBJ PREPOBJ DET DEF NS)	(the cube...)
(DET DEF NPL NS)	the
(NOUN NS)	cube
(CLAUSE RSQ WHRS)*	(...)

Figure 44 -- Syntactic Structure 6

The CLAUSE program is immediately dispatched by the feature WHRS to look for a RELWD. It finds "which", and marks itself as NGREL. It then goes on to look for a (VG NAUX) just as our QUESTION NGQ clause did above. Remember that WH- questions and

WHRS clauses share a great deal of the network, and they share much of the program as well. This time the VG program fails, since the next word is "I", so the CLAUSE program decides that the clause "which I..." is not a SUBJREL. It adds the temporary feature NSUBREL, indicating this negative knowledge, but not deciding yet just what we do have. It then goes to the point in the normal clause program which starts looking for the major constituents of the clause -- subject, verb, etc. We call (PARSE NG SUBJ) and succeed with the PRONG "I". We then look for a VG, and find "wanted". In this case, since the verb is PAST tense, it doesn't need to agree with the subject (only the tenses beginning with PRES show agreement). The feature NAGR marks the non-applicability of agreement. The parsing tree from the WHRS node on down is now:

(CLAUSE RSQ WHRS NGREL NSUBREL)*	(which I wanted...)
(RELWD)	which
(NG SUBJ PRONG NFS)	(I)
(PRON NFS)	I
(VG NAGR (PAST))	(wanted)
(VB MVB PAST TRANS T00BJ SUBT0BJ)	wanted

Figure 45 -- Syntactic Structure 7

The CLAUSE program notes that the MVB is TRANS and begins to look for an OBJ1. This time it also notes that the verb is a T00BJ and a SUBT0BJ (it can take a T0 clause as an object, as in "I wanted to go", or a SUBT0, as in "I wanted you to go." Since

the next word isn't "to", it decides to look for a SUBTO clause, calling (PARSE CLAUSE RSNG OBJ OBJ1 SUBTO). In fact, this checking for different kinds of RSNG clauses is done by a small function named PARSEREL, which looks at the features of the MVB, and calls the appropriate clauses. PARSEREL is used at several points in the grammar, and one of main advantages of writing grammars as programs is that we can write such auxiliary programs (whether in PROGRAMMAR or LISP) to make full use of regularities in the syntax.

The CLAUSE program is called recursively to look for the SUBTO clause "you to pick up". It finds the subject "you", and calls (PARSE VG TO) since it needs a verb group of the "to" type. The VG program notices this feature and finds the appropriate VG (which is again NAGR). The PRT mechanism operates as described in the first example, and the bottom of our structure now looks like:

(CLAUSE RSQ WHRS NGREL NSUBREL)	(which I wanted you to pick up)
(RELWD)	which
(NG SUBJ PRONG NFS)	(I)
(PRON NFS)	I
(VG NAGR (PAST))	(wanted)
(VB MVB PAST TRANS TDOBJ SUBTOBJ)	wanted
(CLAUSE RSNB SUBTO OBJ OBJ1 PRT)*	(you to pick up)
(NG SUBJ PRONG NPL)	(you)
(PRON NPL)	you
(VG TO NAGR)	(to pick)
()	to
(VB MVB INF TRANS VPRT)	pick
(PRT)	up

Figure 46 -- Syntactic Structure 8

Notice that we have a transitive verb-particle combination, "pick up", with no object, and no words left in the sentence. Ordinarily this would cause the program to start backtracking -- checking to see if the MVB is also intransitive, or if there is some way to reparse the clause. However we are in the special circumstance of an embedded clause which is somewhere on the parsing tree below a relative clause with an "unattached" relative. In the clause "which I told you to pick up", "I" is the subject, and the CLAUSE "you to pick up" is the object. The "which" has not been related to anything. There is a small program named UPCHECK which uses PROGRAMMAR's ability to look around on the parsing tree. It looks for this special situation, and when it finds it does three things: 1) Mark the current clause as UPREL, and the appropriate type of UPREL for

the thing it is missing (in this case OBJ1UPREL). 2) Remove the feature NSUBREL from the clause with the unattached relative 3) Replace it with DOWNREL to indicate that the relative has been found below. This can all be done with simple programs using the basic PROGRAMMAR primitives for moving around the tree (see section 2.4.10) and manipulating features at nodes (see 2.4.11). The information which is left in the parsing tree is sufficient for the semantic routines to figure out the exact relationships between the various pieces involved.

In this example, once the CLAUSE "to pick up" has been marked as OBJ1UPREL, it has enough objects, and can return success since the end of the sentence has arrived. The CLAUSE "which I want you to pick up" has an object, and has its relative pronoun matched to something, so it also succeeds, as does the NG "the cube...", the PREPG "by the cube..", and the MAJOR CLAUSE. The final result is shown in Figure 47.

Even in this fairly lengthy description, we have left out much of what was going on. For example we have not mentioned all of the places where the CLAUSE program checked for adverbs (like "usually" or "quickly"), or the VG program looked for "not", etc. These are all "quick" checks, since there is a PROGRAMMAR command which checks the features of the next word. In following the actual programs, the course of the process would be exactly as described, without backups or other attempts to parse major structures.

(CLAUSE MAJOR QUESTION NGQ SUBJQ PASV AGENT)

(NG QUEST DET NUMDET NPL INDEF)	(how many blocks)
(DET QDET NPL INDEF)	how
()	many
(NOUN NPL)	blocks

(VG NAUX VPL PASV (PRES))	(are supported)
(VB AUX BE PRES VPL)	are
(VB MVB EN TRANS)	supported

(PREPG AGENT)	(by the cube which I wanted you to pick up)
(PREP)	by

(NG OBJ PREPOBJ DET DEF NS)	(the cube which I wanted you to pick up)
(DET DEF NPL NS)	the
(NOUN NS)	cube

(CLAUSE RSQ WHRS NGREL DOWNREL TRANS)	(which I wanted you to pick up)
(RELWD)	which

(NG SUBJ PRONG NFS)	(I)
(PRON NFS)	I

(VG NAGR (PAST))	(wanted)
(VB MVB PAST TRANS TOOBJ SUBTOBJ)	wanted

(CLAUSE RSNQ SUBTO OBJ OBJ1 PRT TRANS UPREL OBJ1UPREL)	(you to pick up)
---	------------------

(NG SUBJ PRONG NPL)	(you)
(PRON NPL)	you

(VG TO NAGR)	(to pick)
()	to
(VB MVB INF TRANS VPRT)	pick

(PRT)	up
-------	----

Figure 47 - Syntactic Structure 9
 "How many blocks are supported by the cube
 which I wanted you to pick up?"

This may seem like a quite complex process and complex grammar, compared to other systems, or even our own examples in Section 2.2. This is because language is indeed a highly complex phenomenon. We have tried to handle a great deal more of the complexity of English than any of the previous language-understanding systems. It is only due to the fact that PROGRAMMAR gives us an easy framework in which to include complexity that it was at all possible to include such a detailed grammar as only one part of a project carried out by a single person in less than two years.

2.3.12 Analysis of Word Endings

This section describes the "spelling rules" used by the program in recognizing inflectional endings of words. For spoken language, these would be called the "morphophonemic" rules, but since we deal with written language, they are "morpho-graphemic."

These rules enable a reader to recognize that, for example, "pleasing" is a form of "please", while "beating" is a form of "beat". There is a structure of conventions for doubling consonants, dropping "e", changing "i" to "y", etc. when adding endings, and a corresponding set for removing them.

A word like "running" need not have separate entry in the dictionary, since it is a regular inflected form of "run". The program can use an interpretive procedure to discover the underlying form and attach the appropriate syntactic features for the inflection.

In designing a formalism for these rules, it seems most natural to express them as a program for interpretation. The flow chart in Figure 48 is designed to handle a number of inflectional endings -- "-n't" for negative, "-s" and "-'" for possessive, "-s" and its various forms for plural nouns and singular third-person verbs, "-ing", "-ed", and "-en" verb forms, the superlative "-est" and comparative "-er", and the adverbial "-ly".

As the flowchart shows, these endings share many aspects of

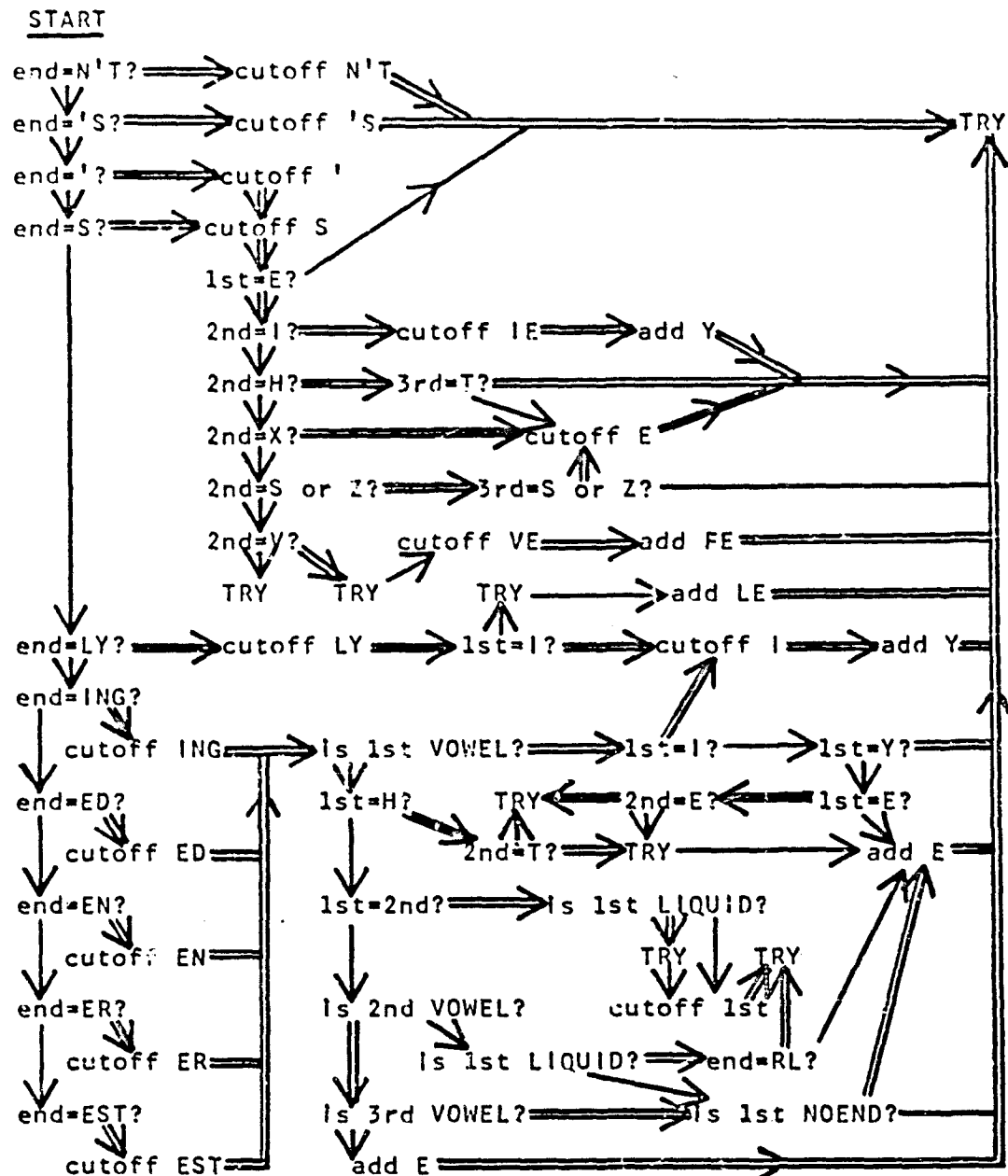


Figure 48 -- Analysis of English Endings

morphographemic structure, and the program representation is able to capture these generalities as well as detailing those aspects peculiar to each ending. It is not a complete description, but covers a wide variety of words, and could easily be expanded to treat more special cases.

The description uses a few simple notations in addition to normal flowchart conventions. "No" and "yes" answers to branch conditions are represented by single and double arrows respectively. The function "cutoff" indicates what is to be cut off of the end of the word. The ordinals "1st", "2nd", etc. count letters from the end of the word backwards, ignoring those which have been cut off. Several classes of letters are relevant to endings -- VOWEL includes (A E I O U Y), LIQUID is (L R S V Z), and NOEND is (C G S V Z). The label "TRY" represents the part of the program which tries looking up the supposed root in the dictionary. It uses whatever letters have not been "cut off". At some places, the program tries an interpretation, then if that fails, carries out a different analysis and tries again.

The program can correctly analyze such roots as: bashes - bash, bathes - bathe, leaning - lean, leaving - leave, dented - dent, danced - dance, dogs - dog, kisses - kiss, curved - curve, curled - curl, rotting - rot, rolling - roll, played - play, plied - ply, realest - real, palest - pale, knives - knife, prettily - pretty, nobly - noble, etc.

It is important to note that exceptions do not need to be explicitly included in the analysis program. The word "was" is directly in the dictionary, and the rules will therefore never be applied in trying to analyze it. The distinction between lexical idiosyncracies and generalities is empirical. A productive rule like the one relating "sang" and "sing", or "rang" and "ring" might well be included in a more complete program, while less productive ones (such as relating "will" to "won't") will be left as separate dictionary entries.

It is tempting to see this program as a simple finite state machine, but this is not the best formalism for several reasons. First, the tests which can be done to a word in deciding on a transition are not, in general, simple checks of the next input letter. Whether a certain analysis is possible may depend, for example, on how many syllables there are in the word, or on some complex phonological calculation involving vowel shifts. Semantic and syntactic information should be applied as well in many cases.

The morphographemic analysis alone is sufficient to do a great deal of the interpretation of English endings. In fact, some systems (<Thorne 1969>) use it to avoid having a dictionary of open class words. The inflection of the words in the input determines their syntactic class possibilities. Of course, without a dictionary, they cannot attempt to deal with the meaning of words, beyond trying to match them with other

occurrences of the same root.

People rarely operate at this level of ignorance. They use their lexical knowledge to realize that "under" is not a comparative form of some adjective "und", and that "bely" is not an adverbial form of "be". This knowledge can readily be integrated into the interpretive program. Once a possible morphographemic analysis is found, the hypothetical root can be checked in the dictionary. If it is not there, the program can try another analysis. Notice that in the flow chart, words ending in a double LIQUID, followed by an inflection, are first tried in the form with the doubled consonant (as in "rolling" - "roll"), then if that fails, another try is made with the single consonant ("patrolled" - "patrol").

If the root is found in the dictionary, its syntactic class is checked to see if the ending is appropriate (e.g. a "-est" ending can apply only to an ADJ). The list of syntactic features is changed to take the ending into account. For example, the ending "-s" changes the feature NS (singular noun) to NPL (plural noun) and the feature INF (infinitive verb) to V3PS (third-person singular verb). A word which fits into more than one category may have more than one set of features changed, in determining all of the possible interpretations.

The problems involved in analyzing more complicated endings, such as the "-tion", "-ment", and "-er" endings of nominalizations will be discussed in a later paper.

2.4 Programming Details

2.4.1 Operation of the System

Since the grammar is itself a program, there is not much overhead mechanism needed for the basic operation of the parser. Instead, the system consists mostly of special functions to be used by the grammar. The system maintains a number of global variables, and keeps track of the parsing tree as it is built by the main function, PARSE. When the function PARSE is called for a UNIT which has been defined as a PROGRAMMAR program, the system collects information about the currently active node, and saves it on a pushdown list. It then sets up the necessary variables to establish a new active node, and passes control to the PROGRAMMAR program for the appropriate unit. If this program succeeds, the system attaches the new node to the tree, and returns control to the node on the top of the PDL. If it fails, it restores the tree to its state before the program was called, then returns control. A PROGRAMMAR program is actually converted by a simple compiler to a LISP program and run in that form. The variables and functions available for writing PROGRAMMAR programs are described in the rest of part 2.4. In order to make these details more independent of our detailed grammar of English, we will continue to use a simplified grammar whenever possible. We use the hypothetical grammar begun in 2.2, and try to use full length feature names for easier understanding.

When the function PARSE is called with a first argument which has not been defined as a PROGRAMMAR program, it checks to see whether the next word has all of the features listed in the arguments. If so, it forms a new node pointing to that word, with a list of features which is the intersection of the list of features for that word with the allowable features for the word class indicated by the first argument of the call. For example, the word "blocks" will have the possibility of being either a plural noun or a third-person-singular present-tense verb. Therefore, before any parsing it will have the features (NOUN VERB N-PL VB-3PS TRANSITIVE PRESENT). If the expression (PARSE VERB TRANSITIVE) is evaluated when "blocks" is the next word in the sentence to be parsed, the feature list of the resulting node will be the intersection of this combined list with the list of allowable features for the word-class VERB. If we have defined:

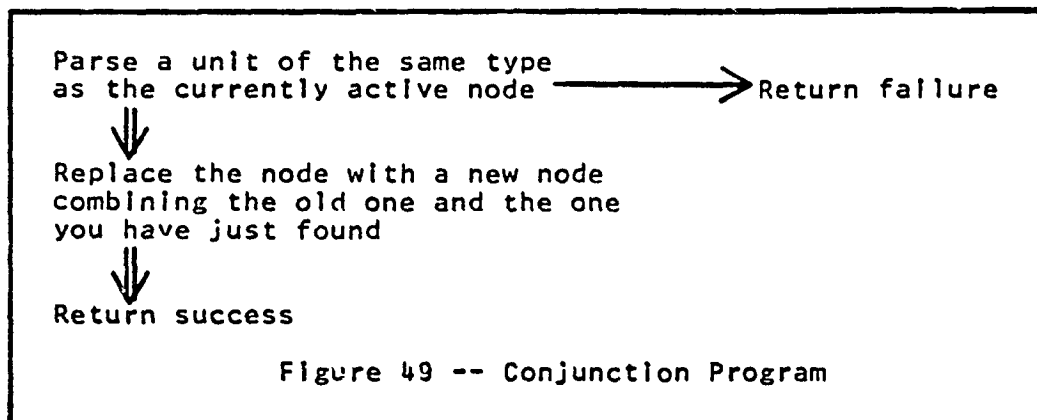
```
(DEFPROP VERB (VERB INTRANSITIVE TRANSITIVE PRESENT PAST
VB-3PS VB-PL) ELIM),
```

the new feature list will be (VERB TRANSITIVE PRESENT VB-3PS). (ELIM is simply a property indicator chosen to indicate this list which ELIMinates features). Thus, even though words may have more than one part of speech, when they appear in the parsing tree, they will exhibit only those features relevant to their actual use in the sentence.

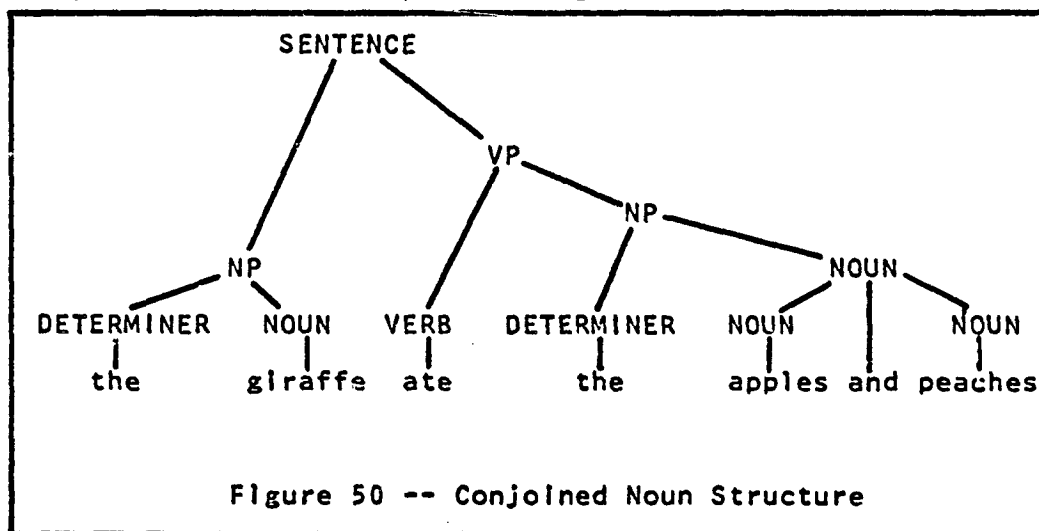
2.4.2 Special Words

Some words must be handled in a very special way in the grammar. The most prevalent are conjunctions, such as "and" and "but". When one of these is encountered, the normal process is interrupted and a special program is called to decide what steps should be taken in the parsing. This is done by giving these words the grammatical features SPEC or SPECL. Whenever the function PARSE is evaluated, before returning it checks the next word in the sentence to see if it has the feature SPEC. If so, the SPEC property on the property list of that word indicates a function to be evaluated before parsing continues. This program can in turn call PROGRAMMAR programs and make an arbitrary number of changes to the parsing tree before returning control to the normal parsing procedure. SPECL has the same effect, but is checked for when the function PARSE is called, rather than before it returns. Various other special variables and functions allow these programs to control the course of the parsing process after they have been evaluated. By using these special words, it is possible to write amazingly simple and efficient programs for some of the aspects of grammar which cause the greatest difficulty. This is possible because the general form of the grammar is a program.

For example, "and" can be defined as a program which is diagrammed:

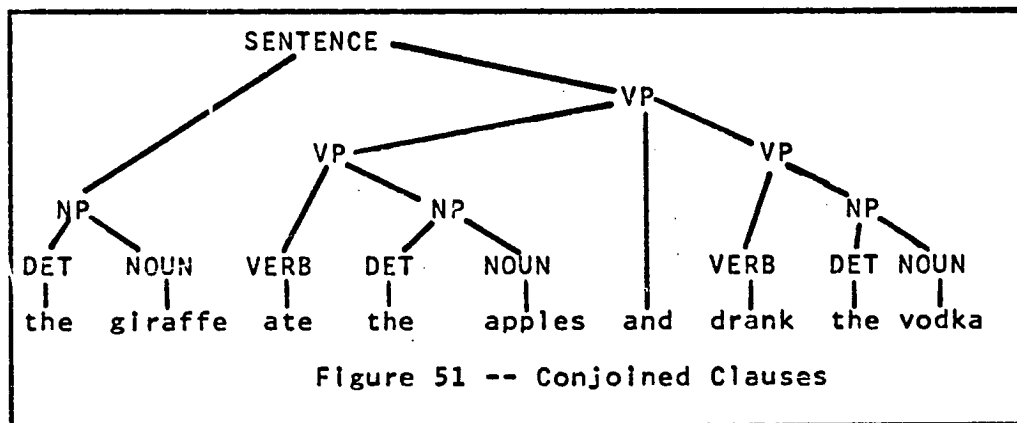


For example, given the sentence "The giraffe ate the apples and peaches." the program would first encounter "and" after parsing the NOUN apples. It would then try to parse a second NOUN, and would succeed, resulting in the structure:



If we had the sentence, "The giraffe ate the apples and drank the vodka." the parser would first try the same thing. However, "drank" is not a NOUN, so the AND program would fail

and the NOUN "apples" would be returned unchanged. This would cause the NP "the apples" to succeed, so the AND program would be called again. It would fail to find a NP beginning with "drank", so the NP "the apples" would be returned, causing the VP to succeed. This time, AND would try to parse a VP and would find "drank the vodka". It would therefore make up a combined VP and cause the entire SENTENCE to be completed with the structure:



The program to actually do this would take only 3 or 4 lines in a PROGRAMMAR grammar. In the actual system, it is more complex as it handles lists (like "A, B, and C") other conjunctions (such as "but") and special constructions (such as "both A and B"). The conjunction program is called by all of the conjunctions, the words "either", "neither", and "both", and the mark ",", which appears as a separate word in the input.

The function ** is used to look ahead for a repetition of the special word, as in "...and...and...". If one is found, a

unit of the type most recently parsed is parsed again, trying to extend all the way to the repeated conjunction or comma. This is iterated as long as there are repetitions, with special checks made for structures like "A, B, and C" or "A and B but not C". As each new node is parsed, its structure is saved, and when the last is found, a new node is created for the compound. Its features combine those for the type of conjunction with those appropriate for the type of unit (e.g. a compound NG connected with "and" is given the feature "plural" (NPL).) The list of constituent structures is put on the tree as a list of subnodes of the conjoined structure, which then replaces the original unit on the parsing tree.

Compounds with a preceding word like "both" are parsed differently, since the word is encountered before any unit has been parsed. In this case it is possible to adopt the more general philosophy of attempting the longest possible unit first. These words have a SPECL definition, so the program is called as the next unit is about to be parsed. The conjunction program looks for the matching conjunction ("and" with "both", "or" with "either", and "nor" with "neither") and tries to parse the unit extending only to the conjunction. If this succeeds, the normal conjunction procedure is followed. If not, some sub-component is the conjoined one, and nothing happens until the parser attempts a sub-unit, when the process is repeated.

A SPECL program can modify the parsing in several ways.

For example it can call the function FLUSHME, which simply removes the word from the input sentence (i.e. it is ignored). It can take arbitrary actions on the current parsing tree, can indicate to PROGRAMMAR that it should SKIP parsing the unit and use instead results provided by the SPECL program, or it can indicate an action to be taken after the normal parsing is DONE. Finally, a SPEC or SPECL program can abort the entire parsing, indicating a response to the user. For example, the word "thank" calls a SPECL program which checks to see if the next word is "you". If so, the parsing is given up, and the system replies "YOU'RE WELCOME". Currently there is no backup procedure to modify the interpretation of an ambiguous structure like "A and B or C". This will in fact be parsed as (A and (B or C)). Notice that "either A and B or C" will be parsed correctly as ((A and B) or C).

The exact format for a SPEC or SPECL definition is a LISP list to which will be appended two items -- the initial feature list of the unit being parsed and an indicator of whether it is a word or a unit which called the program. The resultant form is then EVAL'ed.

2.4.3 Possessives

One of the best examples of the advantages of procedural grammars is the ability to handle left-branching structures like possessives. In a normal top-down parser, these present difficulties, since any NG can begin with a possessive NG, which can in turn begin with a possessive NG, etc., as in "my mother's sister's student's cat's fur". Special care must be taken to avoid infinite loops.

In our grammar this is handled by a check after the NOUN or PRONOUN is found in a NG. If it has the feature "possessive" (POSS) (e.g. "my" or "block's") a node is created for the NG thus far parsed, and this is placed on the tree as a constituent (the determiner) of a NG to be continued. The program then returns to the point where it was after finding a determiner, and continues looking. This can happen any number of times, but in each case is triggered by the presence of another POSSessive word. It therefore loops only as much as necessary. This departure from top-down parsing involves no changes to the parser, and only a simple loop in the program. Any other left-branching structure can be handled similarly.

2.4.4 The Dictionary

Since PROGRAMMAR is embedded in LISP, the facilities of LISP for handling atom names are used directly. To define a word, a list of grammatical features is put on its property list under the indicator WORD, and a semantic definition under the indicator SMNTC. Two facilities are included to avoid having to repeat information for different forms of the same word. First, there is an alternate way of defining words, by using the property indicator WORD1. This indicates that the word given is an inflected form, and its properties are a modified form of the properties of its root. A WORD1 definition has three elements, the root word, the list of features to be added, and the list of features to be removed. For example, we might define the word "go" by: (DEFPROP GO (VERB INTRANSITIVE INFINITIVE) WORD) We could then define "went" as (DEFPROP WENT (GO (PAST)(INFINITIVE)) WORD1) This indicates that the feature INFINITIVE is to be replaced by the feature PAST, but the rest (including the semantic definition) is to remain the same as for "go".

The other facility is an automatic system which checks for simple modifications, such as plurals, "-ing," forms, "-er" and "-est" forms and so forth. If the word as typed in is not defined, the program looks at the way it is spelled, tries to remove its ending (taking into account rules such as changing "running" to "run", but "buzzing" to "buzz"). It then tries to

find a definition for the reduced root word, and if it succeeds, it makes the appropriate changes for the ending (such as changing the feature SINGULAR to PLURAL). The program which does this is not a part of the PROGRAMMAR system but is specifically built for English. It is described in section 2.3.12.

Everything else described in this section is designed generally for the parsing of any language. In any particular language, this input function would have to be written according to the special rules of morphographic structure. The requirement for such a program is that its output must be a list, each member of which corresponds to a word in the original sentence, and is in the form described in section 2.4.8. This list is bound to the variable SENT, and is the way in which PROGRAMMAR sees its input.

The other form of data in the dictionary is tables of verb-particle and preposition-preposition combinations like "pick up" or "on top of". The table is stored on the property list of the initial word under the indicator PRTT or PREPP respectively. It is an association list, each member of which has the second word of the combination as its CAR, and a word parsing structure (see section 2.4.8) as its CADR. There may be more than one combination for the same initial word (e.g. "pick up", "pick out"), and a three-word combination can be defined by making the CADR be an association list of the same form for a third word.

2.4.5 Backup Facilities

As explained in section 2.2.7, there is no automatic backup, but there are a number of special functions which can be used in writing grammars. The simplest, (POPTO X) simply removes nodes from the tree. The argument is a list of features, and the effect is to remove daughters of the currently active node, beginning with the rightmost and working leftward until one is reached with all of those features. (POP X) is the same, except that it also removes the node with the indicated features. If no such node exists, neither function takes any action. (POP) is the same as (POP NIL), and a non-nil value is returned by both functions if any action has been taken.

A very important feature is the CUT variable. One way to do backup is to first try to find the longest possible constituent at any point, then if for any reason an impasse is reached, to return and try again, limiting the constituent from going as far along in the sentence. For example, in the sentence "Was the typewriter sitting on the cake?", the parser will first find the auxiliary verb "was", then try to parse the subject. It will find the noun group "the typewriter sitting on the cake", which in another context might well be the subject ("the typewriter sitting on the cake is broken."). It then tries to find the verb, and discovers none of the sentence is left. To back up, it must change the subject. A very clever program would look at the structure of the noun group and would

realize that the modifying clause "sitting on the cake" must be dropped. A more simple-minded but still effective approach would use the following instructions:

```
(** N PW)
(POP)
((CUT PTW)SUBJECT (ERROR))
```

The first command sets the pointer PTW to the last word in the constituent (in this case, "cake"). The next removes that constituent. The third sets a special pointer, CUT to that location, then sends the program back to the point where it was looking for a subject. It would now try to find a subject again, but would not be allowed to go as far as the word "cake". It might now find "the typewriter sitting," an analog to "The man sitting is my uncle." If there were a good semantic program, it would realize that the verb "sit" cannot be used with an inanimate object without a location specified. This would prevent the constituent "the typewriter sitting" from ever being parsed. Even if this does not happen, the program would fail to find a verb when it looked at the remaining sentence, "on the cake." By going through the cutting loop again, it would find the proper subject, "the typewriter," and would continue through the sentence.

Once a CUT point has been set for any active node, no descendant of that node can extend beyond that point until the CUT is moved. Whenever a PROGRAMMAR program is called, the variable END is set to the current CUT point of the node which

called it. The CUT point for each constituent is initially set to its END. When the function PARSE is called for a word, it first checks to see if the current CUT has been reached (i.e. N and CUT are the same), and if so it fails. The third branch in a three-direction branch statement is taken if the current CUT point has been reached. The CUT pointer is set with the function CUT of one argument.

2.4.6 Auxilliary Functions

Since PROGRAMMAR grammars are programs, they can call subroutines just as any other program. These subroutines can use PROGRAMMAR primitives like PARSE and *, as well as returning values for use in the main program. One example used in our grammar is UPCHECK, used to see if the current node is embedded in a structure which could be an UPREL like "the man I wanted you to see." It is used in conjunction with UPMOD which makes the appropriate changes to the parsing tree. They both use primitives like * to find and change the elements.

In order to simplify the search for rank-shifted clauses, a function PARSEREL was written. It takes as arguments a list of clause types (like REPORT, ING, etc.) a corresponding list of features to look for on the main verb, a pointer to that verb, and the rest of the information to be included in the call to PARSE. PARSEREL then loops through these lists, attempting to parse various types of RSNG clauses if they are in accord with the restrictions associated with the verb and the use of the clause in the sentence. It uses the function PARSE to modify the parsing tree before returning to the main CLAUSE program.

2.4.7 Messages

To write good parsing programs, we may at times want to know why a particular PROGRAMMAR program failed, or why a certain pointer command could not be carried out. In order to facilitate this, two message variables are kept at the top level of the system, MES, and MESP. Messages can be put on MES in two ways, either by using the special failure directions in the branch statements (see section 2.2.5) or by using the functions M and MQ, which are exactly like F and FQ, except they put the indicated feature onto the message list ME for that unit. When a unit returns either failure or success, MES is bound to the current value of ME, so the calling program can receive an arbitrary list of messages for whatever purpose it may want them. MESP always contains the last failure message received from ** or *.

2.4.8 The Form of the Parsing Tree

Each node is actually a list structure with the following information:

FE	the list of features associated with the node
NB	the place in the sentence where the constituent begins
N	the place immediately after the constituent
H	the subtree below that node (actually a list of its daughters in reverse order, so that H points to the last constituent parsed)
SM	a space reserved for semantic information

These symbols can be used in two ways. If evaluated as variables, they will always return the designated information for the currently active node. C is always a pointer to that node. If used as functions of one argument, they give the appropriate values for the node pointed to by that argument; so (NB H) gives the location in the sentence of the first word of the last constituent parsed, while (FE(NB H)) would give the feature list of that word.

Each word in the sentence is actually a list structure containing the 4 items:

FE	as above
SMWORD	the semantic definition of the word
WORD	the word itself (a pointer to an atom)
ROOT	the root of the word (e.g. "run" if the word is "running").

2.4.9 Variables Maintained by the System

There are two types of variables, those bound at the top level, and those which are rebound every time a PROGRAMMAR program is called.

Variables bound at the top level

N	Always points to next word in the sentence to be parsed
SENT	Always points to the entire sentence
PT PTW	Tree and sentence pointers. See Section 2.4.10
MES MESP	List of messages passed up from lower levels. See Section 2.4.7

Special variables bound at each level

C FE NB SM H	See section 2.4.8
NN CUT END	See section 2.4.5. NN always equals (NOT(EQ N CUT))
UNIT	the name of the currently active PROGRAMMAR program
REST	the list of arguments for the call to PARSE (These form the initial feature list for the node, but as other features are added, REST continues to hold only the original ones.)
T1 T2 T3	Three temporary PROG variables for use by the program in any way needed.
MVB	Bound only when a CLAUSE is parsed used as a pointer to the main verb
ME	List of messages to be passed up to next level See Section 2.4.7

2.4.10 Pointers

The system always maintains two pointers, PT to a place on the parsing tree, and PTW to a place in the sentence. These are moved by the functions * and ** respectively, as explained in section 2.2.10. The instructions for PT are:

C	set PT to the currently active node
H	set PT to most recent (rightmost) daughter of C
DL	(down-last) move PT to the rightmost daughter of its current value
DLC	(down-last completed) like DL, except it only moves to nodes which are not on the push-down list of active nodes.
DF	(down-first) like DL, except the leftmost
PV	(previous) move PT to its left-adjacent sister
NX	(next) move PT to its right-adjacent sister
U	(up) move PT to parent node of its current value
N	Move PT to next word in sentence to be parsed

The pointer PTW always points to a place in the sentence. It is moved by the function ** which has the same syntax as *, and the commands:

N	Set PTW to the next word in the sentence
FW	(first-word) set PTW to the first word of the constituent pointed to by PT
LW	(last-word) like FW
AW	(after-word) like FW, but first word after the constituent
NW	(next-word) Set PTW to the next word after its current value
PW	(previous-word) like NW
SFW	(sentence-first-word) set PTW to the first word in the sentence
SLW	(sentence-last-word) like SFW

Since the pointers are bound at the top level, a program which calls others which move the pointers may want to preserve their location. PTW is a simple variable, and can be saved with a SETQ, but PT operates by keeping track of the way it has been

moved, in order to be able to retrace its steps. This is necessary since LISP lists are threaded in only one direction (in this case, from the parent node to its daughters, and from a right sister to its left sister). The return path is bound to the variable PTR, and the command (PTSV X) saves the values of both PT and PTR under the variable X, while (PTRS X) restores both values.

2.4.11 Feature Manipulating

As explained in section 2.2.6, we must be able to attach features to nodes in the tree. The functions F, FQ, and TRNSF are used for putting features onto the current node, while R and RQ remove them. (F A) sets the feature list FE to the union of its current value with the list of features A. (FQ A) adds the single feature A (i.e. it quotes its argument). (TRNSF A B) was explained in Section 2.2.7. R and RQ are inverses of F and FQ. The functions ISX, ISQ, CQ, and NQ are used to examine features. If A points to a node of the tree or word of the sentence, and B points to a feature, (ISX A B) returns non-nil if that node has that feature. (ISQ A B) is equivalent to (IS A (QUOTE B)), (CQ B) is the same as (ISQ C B) (where C always points to the currently active node), and (NQ B) is the same as (ISQ N B) (N always points to the next word in the sentence left to be parsed).

The function NEXTW checks to see if the root of the next word matches the argument. (NEXTW BE) evaluates to non-NIL only if the next word is some form of the verb "be". PUTF and REMF are used to add and remove features from some node other than the current one. They are FEXPRs whose argument is a list of features, which are put on or removed from the node currently pointed to by the pointer PT.

2.5 Comparison with Other Parsers

2.5.1 Older Parsers

When work first began on analyzing natural language with computers, no theories of syntax existed which were explicit enough to be used. The early machine-translation designers were forced to develop their own linguistics as they worked, and they produced rough and ready versions. The parsers were collections of "packaging routines", "inserted structure passes", "labeling subroutines", etc. (see <Garvin>) which evolved gradually as the grammars were expanded to handle more and more complex sentences. They had the same difficulties as any program designed in this way -- as they became more complex it became harder and harder to understand the interactions within them. Making extensions which were intended to deal with a limited anticipated set of inputs tended to make it difficult to extend the system later.

When the machine-translation effort failed, it seemed clear that it had been premature to try handling all of English without a better background of linguistic theory and an understanding of the mathematical properties of grammars. Computer programs for natural language took two separate paths. The first was to ignore traditional syntax entirely, and to use some sort of more general pattern matching process to get information out of sentences. Systems such as STUDENT, SIR, ELIZA, and Semantic Memory made no attempt to do a complete

syntactic analysis of the inputs. They either limited the user to a small set of fixed input forms or limited their understanding to those things they could get while ignoring syntax.

The other approach was to take a simplified subset of English which could be handled by a well-understood form of grammar, such as one of the variations of context-free grammars. There has been much interesting research on the properties of abstract languages and the algorithms needed to parse them. Using this theory, a series of parsing algorithms and representations were developed. For a summary of the computer parsers designed before 1966, see <Bobrow 1964>. A more recent development was Early's context-free parser <Early> which operates in a time proportional to the cube of the length of a sentence.

The problem faced by all of these parsers (including the mammoth Harvard Syntactic Analyzer (<Kuno>)). is that such simple models are not adequate for handling the full complexity of natural language. This is discussed theoretically in <Chomsky 1957> but for our purposes it is more important to note that many aspects which could theoretically be handled would be included only at the expense of gross inefficiency and unnecessary complexity.

Several people attempted to use Chomsky's transformational grammar as the basis for parsers. (see <Petrick> and <Zwicky>) They tried to "unwind" the transformations to reproduce the deep structure of a sentence, which could then be parsed by a context free "base component". It soon became apparent that this was a very difficult task. Although transformational grammar is

theoretically a "neutral" description of language, it is in fact highly biased toward the process of generating sentences rather than interpreting them. Adapting generation rules to use in interpretation is relatively easy for a context-free grammar, but extremely difficult for transformational grammars. (Woods 1969) discusses the problems of "combinatorial explosion" inherent in the inverse transformational process. The transformational parsers have not gone beyond the stage of handling small subsets of English in an inefficient way.

2.5.2 Augmented Transition Networks

In the past two years, three related parsing systems have been developed to deal with the full complexity of natural language. The first was by Thorne, Bratley, and Dewar (<Thorne 1968 and 1969>), and the more recent ones are by Bobrow and Fraser (<Bobrow 1969>) and Woods (<Woods 1969>). The three programs operate in very similar ways, and since Woods' is the most advanced and best documented, we will use it for comparison. In his paper Woods compares his system with the other two.

The basic idea of these parsers is the "augmented transition network". The parser is seen as a transition network much like a finite-state recognizer used for regular languages in automata theory.

The first extension is in allowing the networks to make recursive calls to other networks (or to themselves). The condition for following a particular state transition is not limited to examining a single input symbol. The condition on the arc can be something like "NP" where NP is the name of an initial state of another network. This recursively called NP network then examines the input and operates as a recognizer. If it ever reaches an accepting state, it stops, and parsing continues from the end of the NP arc in the original network. These "recursive transition networks" have the power of a context-free grammar, and the correspondence between a network

and its equivalent grammar is quite simple and direct.

To parse the full range of natural language, we need a critical addition. Instead of using "recursive transition networks" these parsers use "augmented transition networks", which can "make changes in the contents of a set of registers associated with the network, and whose transitions can be conditional on the contents of those registers. (<Woods 1969>). This is done by "adding to each arc of the transition network an arbitrary condition which must be satisfied in order for the arc to be followed, and a set of structure building actions to be executed if the arc is followed."

Augmented transition networks have the power of Turing machines (since they have changeable registers and can transfer control depending on the state of those registers). Clearly they can handle any type of grammar which could possibly be parsed by any machine. The advantages lie in the ways in which these augmented networks are close to the actual operations of language, and give a natural and understandable representation for grammars.

2.5.3 Networks and Programs

How does this type of parser compare with PROGRAMMAR? Is there anything in common between grammars which are networks and grammars which are programs? The reader may have already seen the "joke" in this question. In fact these are just two different ways of talking about doing exactly the same thing!

Picture a flowchart for a PROGRAMMAR grammar, in which calls to the function PARSE are drawn on the arcs rather than at the nodes. Every arc then is either a request to accept the next word in the input (when the argument of PARSE is a word class), or a recursive call to one of the grammar programs. At each node (i.e. segment of program between conditionals and PARSE calls) we have "a set of arbitrary structure building actions." Our flowchart is just like an augmented transition network.

Now picture how Woods' networks are fed to the computer. He uses a notation (see <Woods 1969> p. 17) which looks very much like a LISP-embedded computer language, such as PROGRAMMAR or PLANNER. In fact, the networks could be translated almost directly into PLANNER programs (PLANNER rather than LISP or PROGRAMMAR because of the automatic backup features -- see discussion below).

It is an interesting lesson in computer science to look at Woods' discussion of the advantages of networks, and "translate" them into the advantages of programs. For example, he talks about efficiency of representation. "A major advantage of the transition network model is...the ability to merge the common parts of many context free rules."

Looking at grammars as programs, we can call this "sharing subroutines". He says "The augmented transition network, through its use of flags allows for the merging of similar parts of the network by recording information in registers and interrogating it...and to merge states whose transitions are

similar except for conditions on the contents of the registers." This is the use of subroutines with parameters. In addition, the networks can "capture the regularities of the language...whenever there are two essentially identical parts of the grammar which differ only in that the finite control part of the machine is remembering some piece of information...it is sufficient to explicitly store the distinguishing piece of information in a register and use only a single copy of the subgraph." This is clearly the use of subroutines with an argument!

Similarly we can go through the arguments about efficiency, the ease of mixing semantics with syntax, the ability to include operations which are "natural" to the task of natural language analysis, etc. All of them apply identically whether we are looking at "transition networks" or "programs".

What about "perspicuity"? Woods claims that augmented transition networks retain the perspicuity (ease of reading and understanding by humans) of simpler grammar forms. He says that transformational grammars have the problem that "the effect of a given rule is intimately bound up with its interrelation to other rules...it may require an extremely complex analysis to determine the effect and purpose." (Woods 1969 p.38) This is true, but it would also be true for any grammar complex enough to handle all of natural language. The simple examples of transition networks are indeed easy to read (as are simple

examples of most grammars), but in a network for a complete language, the purpose of a given state would be intimately bound up with its interrelation to other states, and the grammar will not be as "perspicuous" as we might hope. This is just as true for programs, but no more so. If we look at the flow chart instead of the listing, programs are equally perspicuous to networks.

If the basic principles are really the same, are there any differences at all between Woods' system and ours? The answer is yes, they differ not in the theoretical power of the parser, but in the types of analysis being carried out.

The most important difference is the theory of grammar being used. All of the network systems are based on transformational grammar. They try to reproduce the "deep structure" of a sentence while doing surface structure recognition. This is done by using special commands to explicitly build and rearrange the deep structures as the parsing goes along. PROGRAMMAR is oriented towards systemic grammar, with its identification of significant features in the constituents being parsed. It therefore emphasizes the ability to examine the features of constituents anywhere on the parsing tree, and to manipulate the feature descriptions of nodes.

In section 2.1 we discussed the advantages of systemic grammar for a language understanding system. Either type of parser could be adapted to any type of grammar, but PROGRAMMAR was specially designed to include "natural" operations for systemic understanding of sentences.

A second difference is in the implementation of special additions to the basic parser. For example in section 2.4.2 we discussed the way in which words like "and" could be defined to

act as "demons" which interrupt the parsing at whatever point they are encountered, and start a special program for interpreting conjoined structures. This has many uses, both in the standard parts of the grammar (such as "and") and in handling idioms and unusual structures. If we think in network terms, this is like having a separate arc marked "and" leading from every node in the network. Such a feature could probably be added to the network formulation, but it seems much more natural to think in terms of programs and interrupts.

A third difference is the backup mechanism. The network approach assumes some form of nondeterminism. If there are several arcs leaving a node, there must be some way to try following all of them. Either we have to carry forward simultaneous interpretations, or keep track of our choices in such a way that the network can automatically revise its choice if the original choice does not lead to an accepting state. This could be done in the program approach by using a language such as PLANNER with its automatic backup mechanisms. But in section 2.2.7 we discussed the question of whether it is even desirable to do so in handling natural language.

We pointed out the advantage of an intelligent parser which can understand the reasons for its failure at a certain point, and can guide itself accordingly instead of backing up blindly. This is important for efficiency, and Woods is very concerned with ways to modify the networks to avoid unnecessary and

wasteful backup by "making the network more deterministic." (<Woods 1969> p. 45). It might be interesting to explore a compromise solution in which automatic backup facilities existed, but could be turned on and off. We could do this by giving PROGRAMMAR special commands which would cause it to remember the state of the parsing so that later the grammar could ask to back up to that state and try something else. This is an interesting area for further work on PROGRAMMAR.

It is difficult to compare the performance of different parsers since there is no standard grammar or set of test sentences. Bobrow and Woods have not published the results of any experiments with a large grammar, but Thorne has published two papers (<Thorne 1968, 1969>) with a number of sample parsings. Our system, with its current grammar of English has successfully parsed all of these examples. They took from 1 to 5 seconds apiece. Some samples of more complicated parsings done by the system are given in Appendix B.

CHAPTER 3. Inference

3.1 Basic Approach to Meaning

3.1.1 Representing Knowledge

We have described the process of understanding language as a conversion from a string of of sounds or letters to an internal representation of "meaning". In order to do this, a language-understanding system must have some formal way to express its knowledge of a subject, and must be able to represent the "meaning" of a sentence in this formalism. The formalism must be structured in such a way that the system can use its knowledge to make deductions, accept new information, answer questions, and interpret commands. Choosing a form for this information is of central importance to both a practical system and a theory of semantics.

First we must decide what kinds of things are to be represented in the formalism. As a beginning, we would like to be able to represent "objects", "properties," and "relations." Later we will have to show how these can be combined to express more complicated knowledge. We will describe ways to express the meaning of a wide variety of complex sentences.

Using a simple prefix notation, we can represent such facts as "Boise is a city." and "Noah was the father of Jafeth." as:

(CITY BOISE) (FATHER-OF NOAH JAFETH)

Here, BOISE, NOAH, and JAFETH are specific objects, CITY is a property which objects can have, and FATHER-OF is a relation. It is a practical convenience to list properties and relations first, even though this may not follow the natural English order, so we will do so throughout. Notice that properties are in fact special types of relations which deal with only one object. Properties and relations will be dealt with in identical ways throughout the system. In fact, it is not at all obvious which concepts should be considered properties and which relations. For example, "DeGaulle is old." might be expressed as (OLD DEGAULLE) where OLD is a property of objects or as (AGE DEGAULLE OLD), where AGE is a relation between an object and its age. In the second expression, OLD appears in the position of an object, even though it can hardly be construed as a particular object like BOISE or DEGAULLE. This suggests that we might like to let properties or relations themselves have properties and enter into other relations. This has a deep logical consequence which will be discussed in later sections.

In order to avoid confusion, we will need some conventions about notation. Most objects and relationships do not have simple English names, and those that do often share their names with a range of other meanings. The house on the corner by the market doesn't have a proper name like Jafeth, even though it is just as much a unique object. For the internal use of the system, we will give it a unique name by stringing together a

descriptive word and an arbitrary number, then prefixing the result with a colon to remind us it is an object. The house mentioned above might be called :HOUSE374. Properties and relations must also go under an assumed name, since (FLAT X) might mean very different things depending on whether X is a tire or a musical note. We can do the same thing (using a different punctuation mark, #) to represent these two meanings as #FLAT1 and #FLAT2. When the meaning intended is clear, we will omit the numbers, but leave the punctuation marks to remind us that it is a property or relation rather than a specific object. Thus, our facts listed above should be written:

(#CITY :BOISE) (#FATHER-OF :NOAH :JAFETH), and either
 (#OLD :DEGAULLE) or (#AGE :DEGAULLE #OLD).

We are letting properties serve in a dual function -- we can use them to say things about objects (as in "The sky is blue." -- (#BLUE :SKY)) or we can say things about them as if they were objects (as in "Blue is a color." -- (#COLOR #BLUE)). We want to extend this even further, and allow entire relationships to enter into other relationships. (We distinguish between "relation", the abstract symbol such as #FATHER-OF, and "relationship", a particular instance such as (#FATHER-OF :NOAH :JAFETH)). In accord with our earlier convention about naming things, we can give the relationship a name, so that we can treat it like an object and say (#KNOW :REL76) where :REL76 is a name for a particular relationship

like (#FATHER-OF :NOAH :JAFETH). We can keep straight which name goes with which relationship by putting the name directly into the relationship. Our example would then become (#FATHER-OF :NOAH :JAFETH :REL76). There is no special reason to put the name last, except that it makes indexing and reading the statements easier. We can tell that :REL76 is the name of this relation, and not a participant since FATHER-OF relates only two objects. Similarly, we knew that it has to be a participant in the relationship (#KNOW :I :REL76) since #KNOW needs two arguments.

We now have a system which can be used to describe more complicated facts. "Harry slept on the porch after he gave Alice the jewels." would become a set of assertions:

```
(#SLEEP :HARRY :REL1)  (#LOCATION :REL1 :PORCH)
(#GIVE :HARRY :ALICE :JEWELS :REL2)  (#AFTER :REL1 :REL2)
```

This example points out several facts about the notation. The number of participants in a relationship depends on the particular relation, and can vary from 0 to any number. We do not need to give every relationship a name -- it is present only if we want to be able to refer to that relationship elsewhere. This will often be done for events, which are a type of relationship with special properties (such as time and place of occurrence).

3.1.2 Philosophical Considerations

Before going on, let us stop and ask what we are doing. In the preceding paragraphs, we have developed a notation for representing certain kinds of meaning. In doing so we have glibly passed over issues which have troubled philosophers and linguists for thousands of years. Countless treatises and debates have tried to analyze just what it means to be an "object" or a "property", and what logical status a symbol such as #BLUE or #CITY should have. We will not attempt to give a philosophical answer to these questions, but instead take a more pragmatic approach to meaning.

Language is a process of communication between people, and is inextricably enmeshed in the knowledge that those people have about the world. That knowledge is not a neat collection of definitions and axioms, complete, concise and consistent. Rather it is a collection of concepts designed to manipulate ideas. It is in fact incomplete, highly redundant, and often inconsistent. There is no self-contained set of "primitives" from which everything else can be defined. Definitions are circular, with the meaning of each concept depending on the other concepts.

This might seem like a meaningless change -- saying that the meaning of words is represented by the equally mysterious meanings of "concepts" which exist in the speaker's and hearer's minds, but which are open to neither immediate introspection nor

experiment. However, there is a major difference. The structure of concepts which is postulated can be manipulated by a logical system within the computer. The "internal representation" of a sentence is something which the system can obey, answer, or add to its knowledge. It can relate a sentence to other concepts, draw conclusions from it, or store it in a way which makes it useable in further deductions and analysis.

This can be compared to the use of "forces" in physics. We have no way of directly observing a force like gravity, but by postulating its existence, we can write equations describing it, and relate these equations to the physical events involved. Similarly, the "concept" representation of meaning is not intended as a direct picture of something which exists in a person's mind. It is a fiction of the scientist, valid only in that it gives him a way to make sense of data, and predict actual behavior.

The justification for our use of concepts in this system is the way it actually carries out a dialog which simulates in many ways the behavior of a human language user. For a wider field of discourse, it would have to be expanded in its details, and perhaps in some aspects of its overall structure. However the idea is the same -- that we can in fact gain a better understanding of language use by postulating these fictitious concepts and structures, and analyzing the ways in which they interact with language.

The success of such a theory at actually describing language will depend largely on the power and flexibility of the representation used for the concepts. Later sections of this chapter discuss the reasons why PLANNER is particularly well

sulted for this job.

We would like to consider some concepts as "atomic". (i.e. concepts which are considered to have their own meaning rather than being just combinations of other more basic concepts). A property or relation is atomic not because of some special logical status, but because it serves a useful purpose in relation to the other concepts in the speaker's model of the world. For example, the concept #OLD is surely not primitive, since it can be defined in terms of #AGE and number. However, as an atomic property it will often appear in knowledge about people, the way they look, the way they act, etc. Indeed, we could omit it and always express something like "having an age greater than 30", but our model of the world will be simpler and more useful if we have the concept #OLD available instead.

There is no sharp line dividing atomic concepts from non-atomic ones. It would be absurd to have separate atomic concepts for such things as #CITY-OF-POPULATION-23,485 or #PERSON-WEIGHING-BETWEEN-178-AND-181. But it might in fact be useful to distinguish between #BIG-CITY, #TOWN, and #VILLAGE, or between #FAT, and #THIN, since our model may often use these distinctions.

If our "atomic" concepts are not logically primitive, what kind of status do they have? What is their "meaning"? How are they defined? The answer is again relative to the world-model of the speaker. Facts cannot be classified as "those which

define a concept" and "those which describe it." Ask someone to define #PERSON or #JUSTICE, and he will come up with a formula or slogan which is very limited. #JUSTICE is defined in his world-model by a series of examples, experiences, and specific cases. The model is circular, with the meaning of any concept depending on the entire knowledge of the speaker, (not just the kind which would be included in a dictionary). There must be a close similarity between the models held by the speaker and listener, or there could be no communication. If my concept of #DEMOCRACY and yours do not coincide, we may have great difficulty understanding each other's political viewpoints. Fortunately, on simpler things such as #BLUE, #DOG, and #AFTER, there is a pretty good chance that the models will be practically identical. In fact, for simple concepts, we can choose a few primary facts about the concept and use them as a "definition", which corresponds to the traditional dictionary.

Returning to our notation, we see that it is intentionally general, so that our system can deal with concepts as people do. In English we can treat events and relationships as objects, as in "The war destroyed Johnson's rapport with the people." Within our representation of meaning we can similarly treat an event such as #WAR or a relationship of #RAPPORT in the same way we treat objects. We do not draw a sharp philosophical distinction between "specific objects", "properties", "relationships", "events", etc.

3.1.3 Complex Information

We now have a way to store a data base of assertions about particular objects, properties, and relationships. Next, we want to handle more complex information, such as "All canaries are yellow.", or "A thesis is acceptable if either it is long or it contains a persuasive argument." This could be done using a formal language such as the predicate calculus. Basic logical relations such as implies, or, and, there-exists, etc. are represented symbolically, and information is translated into a "formula". Thus we might have:

```
(FORALL (X) (IMPLIES(#CANARY X)(#COLOR X #YELLOW)))
(FORALL (X)(IMPLIES
      (AND (#THESIS X)
        (OR (#LONG X)
          (EXISTS (Y)
            (AND (#PERSUASIVE Y)
              (#ARGUMENT Y)
              (#CONTAINS X Y))))))
      (#ACCEPTABLE X)))
```

Figure 52 -- Predicate Calculus Representation

Several notational conventions are used. First, we need variables so that we can say things about objects without naming particular ones. This is done with the quantifiers FORALL and EXISTS. Second, we need logical relations like AND, OR, NOT, and IMPLIES. Using this formalism, we can represent a question as a formula to be "proved". To ask "Is Sam's thesis acceptable?" we could give the formula (#ACCEPTABLE :SAM-THESIS)

to a theorem prover to prove by manipulating the formulas and assertions in the data base according to the rules of logic. We would need some additional theorems which would allow the theorem prover to prove that a thesis is long, that an argument is acceptable, etc.

In some theoretical sense, predicate calculus formulas could express all of our knowledge, but in a practical sense there is something missing. A person would also have knowledge about how to go about doing the deduction. He would know that he should check the length of the thesis first, since he might be able to save himself the bother of reading it, and that he might even be able to avoid counting the pages if there is a table of contents. In addition to complex information about what must be deduced, he also knows a lot of hints and "heuristics" telling how to do it better for the particular subject being discussed.

Most "theorem-proving" systems do not have any way to include this additional intelligence. Instead, they are limited to a kind of "working in the dark". A uniform proof procedure gropes its way through the collection of theorems and assertions, according to some general procedure which does not depend on the subject matter. It tries to combine any facts which might be relevant, working from the bottom up. In our example given above, we might have a very complex theorem for deciding whether an argument is persuasive. A uniform proof procedure might spend a great deal of time checking the persuasiveness of every argument it knew about, since a clause of the form (PERSUASIVE X) might be relevant to the proof. What we would prefer is a way for a theorem to guide the process of deduction in an intelligent way. Carl Hewitt has worked with

this problem and has developed a theorem-proving language called PLANNER <Hewitt 1968, 1969>. In PLANNER, theorems are in the form of programs, which describe how to go about proving a goal, or how to deduce consequences from an assertion. This is described at length in section 3.3, and forms the basis for the inference part of our English understander. In PLANNER, our sentence about thesis evaluation could be represented as shown in Figure 53.

This is similar in structure to the predicate calculus representation given above, but there are important differences. The theorem is a program, where each logical operator indicates a definite series of steps to be carried out. THGOAL says to try to find an assertion in the data base, or to prove it using other theorems. THUSE gives advice on what other theorems to use, and in what order. THAND and THOR are equivalent to the logical AND and OR except that they give a specific order in which things should be tried. (The "lisping" is to differentiate PLANNER names from the standard LISP functions AND and OR. This same convention is used for all functions which have LISP analogs.)

The theorem EVALUATE says that if we ever want to prove that a thesis is acceptable, we should first make sure it is a thesis by looking in the data base. Next, we should try to prove that it is long, first by using the theorem CONTENTS-CHECK (which would check the table of contents), and if that fails, by

```

(DEFINE THEOREM EVALUATE
    ;EVALUATE is the name we are
    ;giving to the theorem

    (THCONSE(X Y)
        ;this indicates the type of
        ;theorem and names its
        ;variables

    (THGOAL(#THESIS $?X))
        ;show that X is a thesis
        ;the "$?" indicates a variable

    (THOR
        ;THOR is like "or", trying things
        ;in the order given until one works

    (THGOAL(#LONG $?X)(THUSE CONTENTS-CHECK COUNTPAGES))
        ;THUSE says to try the theorem
        ;named CONTENTS-CHECK first,
        ;then if that doesn't work, try
        ;the one named COUNTPAGES

    (THAND
        ;THAND is like "and"

    (THGOAL(#CONTAINS $?X $?Y))
        ;find something Y which is
        ;contained in X

    (THGOAL(#ARGUMENT $?Y))
        ;show that it is an argument

    (THGOAL(#PERSUASIVE $?Y)(THTBF THTRUE))))))
        ;prove that it is persuasive, using
        ;any theorems which are applicable

```

Figure 53 -- ANNER Representation

using a theorem named COUNTPAGES (which might in fact call a simple LISP program which thumbs through the paper.) If they both fail, then we look in the data base for something contained in the thesis, check that it is an argument, and then finally try to prove that it is persuasive. Here, we have used (THTBF THTRUE), which is PLANNER'S way of saying "try anything you know which can help prove it". PLANNER must then go searching through all of its theorems on persuasiveness, just as any other theorem prover would. There are two important changes, though. First, we never need to look at persuasiveness at all if we are able to determine that the thesis is long. Second, we only look at the persuasiveness of arguments which we already know are a part of the thesis. We do not get sidetracked into looking at the persuasiveness theorems except for the cases we really want.

PLANNER also does a number of other things, like maintaining a dynamic data base (assertions can be added or removed to reflect the way the world changes in the course of time), allowing us to control how much deduction will be done when new facts are added to the data base, etc. These are all discussed in section 3.3.

3.1.4 Questions, Statements, and Commands

PLANNER is particularly convenient for a language-understanding system, since it can express statements, commands, and questions directly. We have already shown how assertions can be stated in simple PLANNER format. Commands and questions are also easily expressed. Since a theorem is written in the form of a procedure, we can let steps of that procedure actually be actions to be taken by a robot. The command "Pick up the block and put it in the box." could be expressed as a PLANNER program:

```
(THAND(THGOAL(#PICKUP :BLOCK23))
      (THGOAL(#PUTIN :BLOCK23 :BOX7)))
```

Remember that the prefix ":" and the number indicate a specific object. The theorems for #PICKUP and #PUTIN would also be programs, describing the sequence of steps to be done.

Earlier we asked about Sam's thesis in predicate calculus. In PLANNER we can ask:

```
(THGOAL (#ACCEPTABLE :SAM-THESIS)(THUSE EVALUATE))
```

Here we have specified that our theorem EVALUATE is to be used. If we evaluated this PLANNER statement, the theorem would be called, and executed just as described on the previous pages. PLANNER would return one of the values "T" or "NIL" depending on whether the statement is true or false.

For a question like "What nations have never fought a war?" PLANNER has the function THFIND. We would ask:

```

(THFIND ALL $?X (X Y)
  (THGOAL(#NATION $?X))
  (THNOT
    (THAND(THGOAL(#WAR $?Y))
      (THGOAL(#PARTICIPATED $?X $?Y)))))

```

and PLANNER would return a list of all such countries. Using our conventions for giving names to relations and events, we could even ask:

```

(THFIND ALL $?X (X Y Z EVENT)
  (THGOAL(#CHICKEN $?Y))
  (THGOAL(#ROAD $?Z))
  (THGOAL(#CROSS $?Y $?Z $?EVENT))
  (THGOAL(#CAUSE $?X $?EVENT)))

```

This brief description has explained the basic concepts underlying the deductive part of our language understanding program. To go with it, we need a complex model of the subject being discussed. This is described in section 3.4. Section 3.3 gives more details about the PLANNER language and its uses.

3.2 Comparison with Previous Programs

In Section 3.1 we discussed ways of representing information and meaning within a language-comprehending system. In order to compare our ideas with those in previous systems, we will establish a broad classification of the field. Of course, no set of pigeon-holes can completely characterize the differences between programs, but they can give us some viewpoints from which to analyze different people's work, and can help us see past the superficial differences. We will deal only with the ways that programs represent their information about the subject matter they discuss. Issues such as parsing and semantic analysis techniques are discussed in other sections. We will distinguish four basic types of systems called "special format", "text based", "restricted logic", and "general deductive".

3.2.1 Special Format Systems

Most of the early language understanding programs were of the special format type. Such systems usually use two special formats designed for their particular subject matter -- one for representing the knowledge they keep stored away, and the other for the meaning of the English input. Some examples are: BASEBALL <P.F.Green>, which stored tables of baseball results and interpreted questions as "specification lists" requesting data from those tables; SAD SAM <Lindsay>, which interpreted sentences as simple relationship facts about people, and stored these in a network structure; STUDENT <Bobrow 1964>, which interpreted sentences as linear equations and could store other linear equations and manipulate them to solve algebra problems; and ELIZA <Weizenbaum 1966>, whose internal knowledge is a set of sentence rearrangements and key words, and which sees input as a simple string of words.

These programs all make the assumption that the only relevant information in a sentence is that which fits their particular format. Although they may have very sophisticated mechanisms for using this information (as in CARPS <Charniak>, which can solve word problems in calculus), they are each built for a special purpose, and do not handle information with the flexibility which would allow them to be adapted to other uses. Nevertheless, their restricted domain often allows them to use very clever tricks, which achieve impressive results with a

minimum of concern for the complexities of language.

3.2.2 Text Based Systems

Some researchers were not satisfied with the limitations inherent in the special-format approach. They wanted systems which were not limited by their construction to a particular specialized field. Instead they used English text, with all of its generality and diversity, as a basis for storing information. In these "text based" systems, a body of text is stored directly, under some sort of indexing scheme. An English sentence input to the understander is interpreted as a request to retrieve a relevant sentence or group of sentences from the text. Various ingenious methods were used to find possibly relevant sentences and decide which were most likely to satisfy the request.

PROTOSYNTHESIS I (Simmons 1966) had an index specifying all the places where each "content word" was found in the text. It tried to find the sentences which had the most words in common with the request (using a special weighting formula), then did some syntactic analysis to see whether the words in common were in the right grammatical relationship to each other. Semantic Memory (Quillian 1966) stored a slightly processed version of English dictionary definitions in which multiple-meaning words were eliminated by having humans indicate the correct interpretation. It then used an associative indexing scheme which enabled the system to follow a chain of index references. An input request was in the form of two words instead of a

sentence. The response was the shortest chain which connected them through the associative index (e.g. if there is a definition containing the words A and B and one containing B and C, a request to relate A and C will return both sentences).

Even with complex indexing schemes, the text based approach has a basic problem. It can only spout back specific sentences which have been stored away, and can not answer any question which demands that something be deduced from more than one piece of information. In addition, its responses often depend on the exact way the text and questions are stated in English, rather than dealing with the underlying meaning.

3.2.3 Limited Logic Systems

The "limited logic" approach attempted to correct these faults of text based systems, and has been used for most of the more recent language understanding programs. First, some sort of more formal notation is substituted for the actual English sentences in the base of stored knowledge. This notation may take many different forms, such as "description lists" (Raphael 1964), "kernels" (Simmons 1968) "concept-relation-concept triples" (Simmons 1969), "data nodes" (Quillian 1969), "rings" (Thompson), "relational operators" (Tharp), etc. Each of these forms is designed for efficient use in a particular system, but at heart they are all doing the same thing -- providing a notation for simple assertions of the sort described in section 3.1.1. It is relatively unimportant which special form is chosen. All of the different methods can provide a uniform formalism which frees simple information from being tied down to a specific way of expressing it in English. Once this is done, a system must have a way of translating from the English input sentences into this internal assertion format, and the greatest bulk of the effort in language understanding systems has been this "semantic analysis". We will discuss it at length in chapter 4. For now we are more interested in what can be done with the assertions once they have been put into the desired form.

Some systems (see (Quillian 1969), (Tharp)) remain close to

text based systems, only partially breaking down the initial text input. The text is processed by some sort of dependency analysis and left in a network form, either emphasizing semantic relationships or remaining closer to the syntactic dependency analysis. What is common to these systems is that they do not attempt to answer questions from the stored information. As with text based systems, they try to answer by giving back bits of information directly from the data base. They may have clever ways to decide what parts of the data are relevant to a request, but they do not try to break the question down and answer it by logical inference. Because of this, they suffer the same deficiencies as text based systems. They have a mass of information stored away, but little way to use it except to print it back out.

Most of the systems which have been developed recently fit more comfortably under the classification "limited logic". In addition to their data base of assertions (whatever they are called), they have some mechanism for accepting more complex information, and using it to deduce the answers to more complex questions. By "complex information" we mean the type of knowledge described in section 3.1.3. This includes knowledge containing logical quantifiers and relationships (such as "Every canary is either yellow or purple," or "If A is a part of B and B is a part of C, then A is a part of C."). By "complex questions", we mean questions which are not answerable by giving

out one of the data base assertions, but demand some logical inference to produce an answer.

One of the earliest limited logic programs was SIR (Raphael 1964), which could answer questions using simple logical relations (like the "part" example in the previous paragraph). The complex information was not expressed as data, but was built directly into the SIR operating program. This meant that the types of complex information it could use were highly limited, and could not be easily changed or expanded. The complex questions it could answer were similar to those in many later limited logic systems, consisting of four basic types. The simplest is a question which translates into a single assertion to be verified or falsified (e.g. "Is John a bagel?") The second is an assertion in which one part is left undetermined (e.g. "Who is a bagel?") and the system responds by "filling in the blank". The third type is an extension of this, which asks for all possible blank-fillers (e.g. "Name all bagels."), and the fourth adds counting to this listing facility to answer count questions (e.g. "How many bagels are there?"). SIR had special logic for answering "how many" questions, using information like "A hand has 5 fingers.", and in a similar way each limited logic system had special built-in mechanisms to answer certain types of questions.

The DEACON system (Thompson) had special "verb tables" to handle time questions, and a bottom-up analysis method which

allowed questions to be nested. For example, the question "Who is the commander of the battalion at Fort Fubar?" was handled by first internally answering the question "What battalion is at Fort Fubar?" The answer was then substituted directly into the original question to make it "Who is the commander of the 69th battalion?", which the system then answered. PROTOSYNTHESIS II <Simmons 1968> had special logic for taking advantage of the transitivity of "is" (e.g. "A boy is a person.", "A person is an animal." therefore "A..."). PROTOSYNTHESIS III <Simmons 1969> and SAMENLAQ II <Shapiro> bootstrapped their way out of first-order logic by allowing simple assertions about relationships (e.g. "North-of is the converse of South-of."). CONVERSE <Kellogg> converted questions into a "query language" which allowed the form of the question to be more complex but used simple table lookup for finding the answers.

All of the limited logic systems are basically similar, in that complex information is not part of the data, but is built into the system programs. Those systems which could add to their initial data base by accepting English sentences could accept only simple assertions as input. The questions could not involve complex quantified relationships (e.g. "Is there a country which is smaller than every U.S. state?").

3.2.4 General Deductive Systems

The problems of limited logic systems were recognized very early (see <Raphael 1964> p. 90), and people looked for a more general approach to storing and using complex information. If the knowledge could be expressed in some standard mathematical notation (such as the predicate calculus), then all of the work logicians have done on theorem proving could be utilized to make an efficient deductive system. By expressing a question as a theorem to be proved (see section 3.1.3), the theorem prover could actually deduce the information needed to answer any question which could be expressed in the formalism. Complex information not easily useable in limited logic systems could be neatly expressed in the predicate calculus, and a body of work already existed on computer theorem proving. This led to the "general deductive" approach to language understanding programs.

The early programs used logical systems less powerful than the full predicate calculus (see <Bar-Hillel>, <Coles 1968>, and <Darlington>) but the big boost to theorem proving research was the development of the Robinson resolution algorithm <Robinson>, a very simple "complete uniform proof procedure" for the first order predicate calculus. This meant that it became easy to write an automatic theorem proving program with two important characteristics. First, the procedure is "uniform" -- we need not (and in fact, cannot) tell it how to go about proving things in a way suited to particular subject matter. It has its own

fixed procedure for building proofs, and we can only change the sets of logical statements (or "axioms") for it to work on. Second, it guarantees that if any proof is possible using the rules of predicate calculus, the procedure will eventually find it (even though it may take a very long time). These are very pretty properties for an abstract deductive system, but the question we must ask is whether their theoretical beauty is worth paying the price of low practicality. We would like to argue that in fact they have led to the worst deficiencies of the theorem-proving question-answerers, and that a very different approach is called for.

The "uniform procedure" approach was adopted by a number of systems (see <Green 1968, 1969>) as an alternative to the kind of specialized limited logic discussed in the previous section. It was felt that there must be a way to present complex information as data rather than embedding it into the inner workings of the language understanding system. There are many benefits in having a uniform notation for representing problems and knowledge in a way which does not depend on the quirks of the particular program which will interpret them. It enables a user to describe a body of knowledge to the computer in a "neutral" way without knowing the details of the question-answering system, and guarantees that the system will be applicable to any subject, rather than being specialized to handle only one.

Predicate calculus seemed to be a good uniform notation, but in fact it has a serious deficiency. By putting complex information into a "neutral" logical formula, these systems ignored the fact that an important part of a person's knowledge concerns how to go about figuring things out. Our heads don't contain neat sets of logical axioms from which we can deduce everything through a "proof procedure". Instead we have a large set of heuristics and procedures for solving problems at different levels of generality. Of course, there is no reason why a computer should do things the way a person does, but in ignoring this type of knowledge, programs run into tremendous problems of efficiency. As soon as a "uniform procedure" theorem prover gets a large set of axioms (even well below the number needed for really understanding language), it becomes bogged down in searching for a proof, since there is no easy way to guide its search according to the subject matter. In addition, a proof which takes many steps (even if they are in a sequence which can be easily predicted by the nature of the theorem) may take impossibly long since it is very difficult to describe the correct proving procedure to the system.

It is possible to write theorems in a clever way in order to implicitly guide the deduction process, and a recent paper <Green 1969> describes some of the problems in "techniques for "programming" in first-order logic". First order logic is a declarative rather than imperative language, and to get an imperative effect (i.e. telling it how to go about doing something) takes a good deal of careful thought and clever trickery.

It might be possible to add strategy information to a predicate calculus theorem prover, but with current systems such as QA3, "To change strategies in the current version, the user must know about set-of-support and other program parameters such as level bound and term-depth bound. To radically change the strategy, the user presently has to know the LISP language and must be able to modify certain strategy sections of the

program." (<Green 1969> p.236). In newer programs such as QA4, there will be a special strategy language to go along with the theorem-proving mechanisms. It will be interesting to see how close these new strategy languages are to PLANNER, and whether there is any advantage to be gained by putting them in a hybrid with a resolution-based system. As to the completeness argument, there are good reasons not to have a complete system - these are discussed later in this section.

3.2.5 Procedural Deductive Systems

The problem with the limited logic systems wasn't the fact that they expressed their complex information in the form of programs or procedures. The problem was that these programs were organized in such a way that "...each change in a subprogram may affect more of the other subprograms. The structure grows more awkward and difficult to generalize...Finally the system may become too unwieldy for further experimentation." (<Raphael 1964> p.91). Nevertheless, it was necessary to build in more and more of these subprograms in order to accept new subject matter.

What was needed was the development of new programming techniques so that systems could retain the capability of using procedural information, but at the same time express this information in a simple and straightforward way which did not depend on the peculiarities and special structure of a particular program or subject of discussion.

A system which partially fits this description is Woods' <Woods 1968>. It uses a quantificational query language for expressing questions, then assumes that there are "semantic primitives" in the form of LISP subroutines which decide such predicates as (CONNECT FLIGHT-23 BOSTON CHICAGO) and which evaluate functions such as "number of stops", "owner", etc. The thing which makes this system different from the limited logic systems is that the entire system was designed without reference

to the way the particular "primitive" functions would operate on the data base. In a way, this is avoiding the issue, since the information which the system was designed to handle (the Official Airline Guide) is particularly amenable to simple table-lookup routines. If we had to handle less structured information of the type usually done with theorem provers, these primitive routines might indeed run into the same problems of interconnectedness described in the quote above, and would become harder and harder to generalize.

PLANNER was designed by Carl Hewitt as a goal-oriented procedural language to deal with these problems. It has special mechanisms for dealing with assertions in an efficient way, and in addition has the capability to include any complex information which can be expressed in the predicate calculus. More important, the complex information is expressed in the form of procedures, which can include all sorts of knowledge of how to best go about proving things. The language is "goal-oriented", in that we do not have to be concerned about the details of interaction between the different procedures. If at different places in our knowledge we have theorems which ask whether an object is sturdy (for example in a theorem about support, about building houses, etc.) they are not forced to specify the program which will serve as sturdiness-inspector. Instead they say something like "Try to find an assertion that X is sturdy, or prove it using anything you can." If we know of special

procedures which seem most likely to give a quick answer, we can specify that these should be tried first. But if at some point we add a new sturdiness-tester, we do not need to find out which theorems use it. We need only add it to the data base, and the system will automatically try it (along with any other sturdiness-testers) whenever any theorem gives the go-ahead.

The ability to add new theorems without relating them to other theorems is the advantage of a "uniform" notation. In fact PLANNER is a uniform notation for expressing procedural knowledge just as predicate calculus is a notation for a more limited range of information. The advantage is that PLANNER has a hierarchical control structure. In addition to specifying logical relationships, a theorem can take over control of the deduction process.

We can have complete control over how the system will operate. In any theorem, we can tell it to try to prove a subgoal using only certain theorems (if we know that the goal is bound to fail unless one of them works), we can tell it to try things in a certain order (and the choice of this order can depend on arbitrarily complex calculations which take place when the subgoal is set up) or we can even write a "spoiler" theorem, which can tell the system that a goal is certain to fail, and that no other theorems should even be tried.

Notice that this control structure makes it very difficult to characterize the abstract logical properties of PLANNER, such as consistency and completeness. It is worth pointing out here that completeness may in fact be a bad property. It means (we believe, necessarily) that if the theorem-prover is given something to prove which is in fact false, it will exhaust every

possible way of trying to prove it. By forsaking completeness, we allow ourselves to use good sense in deciding when to give up.

In a truly uniform system, the theorem prover is forced to "rediscover the world" every time it answers a question. Every goal forces it to start from scratch, looking at all of the theorems in the data base (perhaps using some subject-matter-free heuristics to make a rough selection). Because it does not want to be limited to domain-dependent information, it cannot use it at all. PLANNER can operate in this "blindman" mode if we ask it to (and it is less efficient at doing so than a procedure specially invented to operate this way), but it should have to do this only rarely -- when discovering something which was not known or understood when the basic theorems were written. The rest of the time it can go about proving things which it knows how to do, without a tremendous overhead of having to piece together a proof from scratch each time. As mentioned above, it might be possible to patch "strategy programs" onto theorems in conventional theorem-provers in order to accomplish the same goal. In PLANNER we have the advantage that this can be done naturally using the notation, and the strategy is embedded in the PLANNER theorems, which themselves can be looked at as data. In an advanced system a PLANNER program could be written to learn from experience. Once the "blindman mode" finds a proof, the method it used could be

remembered and tried first when a similar goal is generated again. See section 5.1 for more discussion of learning.

To those accustomed to uniform proof procedures, this all sounds like cheating. Is the system really proving anything if you are giving it clues about what to do? Why is it different from a simple set of programmed LISP procedures like those envisioned by Woods? First, the language is designed so that theorems can be written independently of each other, without worrying about when they will be called, or what other theorems and data they will need to prove their subgoals.

The language is designed so that if we want, we can write theorems in a form which is almost identical to the predicate calculus, so we have the benefits of a uniform system. On the other hand, we have the capability to add as much subject-dependent knowledge as we want, telling theorems about other theorems and proof procedures. The system has an automatic goal-tree backup system, so that even when we are specifying a particular order in which to do things, we may not know how the system will go about doing them. It will be able to follow our suggestions and try many different theorems to establish a goal, backing up and trying another automatically if one of them leads to a failure (see section 3.3).

In summary, the main advance in a deductive system using PLANNER is in allowing ourselves to have a data base of procedures rather than formulas to express complex information. This combines the generality and power of a theorem prover with the ability to accept procedural knowledge and heuristics relevant to the data. It provides a flexible and powerful tool to serve as the basis for a language understanding system. The rest of this chapter describes the PLANNER language and the way it is used in our system.

3.3 Programming in PLANNER

3.3.1 Basic Operation of PLANNER

The easiest way to understand PLANNER is to watch how it works, so in this section we will present a few simple examples and explain the use of some of its most elementary features.

First we will take the most venerable of traditional deductions:

```
Turing is a human
All humans are fallible
so
  Turing is fallible.
```

It is easy enough to see how this could be expressed in the usual logical notation and handled by a uniform proof procedure. Instead, let us express it in one possible way to PLANNER by saying:

```
(THASSERT (HUMAN TURING))
      ;This asserts that Turing is human.
(DEFPROP THEOREM1
  (THCONSE (X) (FALLIBLE $?X)
    (THGOAL (HUMAN $?X)))
  THEOREM)
      ;This is one way of saying that all humans
      ;are fallible.
```

The proof would be generated by asking PLANNER to evaluate the expression:

```
(THGOAL (FALLIBLE TURING) (THTBF THTRUE))
```

We immediately see several points. First, there are two different ways of storing information. Simple assertions are stored in a data base of assertions, while more complex sentences containing quantifiers or logical connectives are

expressed in the form of theorems.

Second, one of the most important points about PLANNER is that it is an evaluator for statements written in a programming language. It accepts input in the form of expressions written in the PLANNER language, and evaluates them, producing a value and side effects. THASSERT is a function which, when evaluated, stores its argument in the data base of assertions or the data base of theorems (which are cross-referenced in various ways to give the system efficient look-up capabilities). A theorem is defined with DEFPROP as are functions in LISP.

In this example we have defined a theorem of the THCONSE type (THCONSE means consequent; we will see other types later). This states that if we ever want to establish a goal of the form (FALLIBLE \$?X), we can do this by accomplishing the goal (HUMAN \$?X), where X is a variable. The strange prefix characters are part of PLANNER's pattern matching capabilities. If we ask PLANNER to prove a goal of the form (A X), there is no obvious way of knowing whether A and X are constants (like TURING and HUMAN in the example) or variables. LISP solves this problem by using the function QUOTE to indicate constants. In pattern matching this is inconvenient and makes most patterns much bulkier and more difficult to read. Instead, PLANNER uses the opposite convention -- a constant is represented by the atom itself, while a variable must be indicated by adding an appropriate prefix. This prefix differs according to the exact

use of the variable in the pattern, but for the time being let us just accept \$? as a prefix indicating a variable. The definition of the theorem indicates that it has one variable, X, by the (X) following THCONSE.

The third statement illustrates the function THGOAL, which calls the PLANNER interpreter to try to prove an assertion. This can function in several ways. If we had asked PLANNER to evaluate (THGOAL (HUMAN TURING)) it would have found the requested assertion immediately in the data base and succeeded (returning as its value some indicator that it had succeeded). However, (FALLIBLE TURING) has not been asserted, so we must resort to theorems to prove it.

Later we will see that a THGOAL statement can give PLANNER various kinds of advice on which theorems are applicable to the goal and should be tried. For the moment, (THTBF THTRUE) is advice that causes the evaluator to try all theorems whose consequent is of a form which matches the goal. (i.e. a theorem with a consequent (\$?Z TURING) would be tried, but one of the form (HAPPY \$?Z) or (FALLIBLE \$?Y \$?Z) would not. Assertions can have an arbitrary list structure for their format -- they are not limited to two-member lists or three-member lists as in these examples.) The theorem we have just defined would be found, and in trying it, the match of the consequent to the goal would cause the variable \$?X to be assigned to the constant TURING. Therefore, the theorem sets up a new goal (HUMAN

TURING) and this succeeds immediately since it is in the data base. In general, the success of a theorem will depend on evaluating a PLANNER program of arbitrary complexity. In this case it contains only a single THGOAL statement, so its success causes the entire theorem to succeed, and the goal (FALLIBLE TURING) is proved.

Consider the question "Is anything fallible?", or in logic, (EXISTS (Y)(FALLIBLE Y)). This requires a variable and it could be expressed in PLANNER as:

```
(THPROG (Y) (THGOAL (FALLIBLE $Y)(THTBF THTRUE)))
```

Notice that THPROG (PLANNER's equivalent of a LISP PROG, complete with GO statements, tags, RETURN, etc.) acts as an existential quantifier. It provides a binding-place for the variable Y, but does not initialize it -- it leaves it in a state particularly marked as unassigned. To answer the question, we ask PLANNER to evaluate the entire THPROG expression above. To do this it starts by evaluating the THGOAL expression. This searches the data base for an assertion of the form (FALLIBLE \$Y) and fails. It then looks for a theorem with a consequent of that form, since the recommendation (THTBF THTRUE) says to look at all possible theorems which might be applicable. When the theorem defined above is called, the variable X in the theorem is identified with the variable Y in the goal, but since Y has no value yet, X does not receive a value. The theorem then sets up the goal (HUMAN \$X) with X as

a variable. The data-base searching mechanism takes this as a command to look for any assertion which matches that pattern (i.e. an instantiation), and finds the assertion (HUMAN TURING). This causes X (and therefore Y) to be assigned to the constant TURING, and the theorem succeeds, completing the proof and returning the value (FALLIBLE TURING).

3.3.2 Backup

There seems to be something missing. So far, the data base has contained only the relevant objects, and therefore PLANNER has found the right assertions immediately. Consider the problem we would get if we added new information by evaluating the statements:

```
(THASSERT (HUMAN SOCRATES))
(THASSERT (GREEK SOCRATES))
```

Our data base now contains the assertions:

```
(HUMAN TURING)
(HUMAN SOCRATES)
(GREEK SOCRATES)
```

and the theorem:

```
(THCONSE (X) (FALLIBLE $?X)
           (THGOAL (HUMAN $?X)))
```

What if we now ask, "Is there a fallible Greek?" In PLANNER we would do this by evaluating the expression:

```
(THPROG (X) (THGOAL (FALLIBLE $?X)(THTBF THTRUE))
         (THGOAL (GREEK $?X)))
```

THPROG acts like an AND, insisting that all of its terms are satisfied before the THPROG is happy. Notice what might happen. The first THGOAL may be satisfied by the exact same deduction as before, since we have not removed information. If the data-base searcher happens to run into TURING before it finds SOCRATES, the goal (HUMAN \$?X) will succeed, assigning \$?X to TURING. After (FALLIBLE \$?X) succeeds, the THPROG will then establish the new goal (GREEK TURING), which is doomed to fail since it has not been asserted, and there are no applicable

theorems. If we think in LISP terms, this is a serious problem, since the evaluation of the first THGOAL has been completed before the second one is called, and the "push-down list" now contains only the THPROG. If we try to go back to the beginning and start over, it will again find TURING and so on, ad infinitum.

One of the most important features of the PLANNER language is that backup in case of failure is always possible, and moreover this backup can go to the last place where a decision of any sort was made. Here, the decision was to pick a particular assertion from the data base to match a goal. Other decisions might be the choice of a theorem to satisfy a goal, or a decision of other types found in more complex PLANNER functions such as THOR (the equivalent of LISP OR). PLANNER keeps enough information to change any decision and send evaluation back down a new path.

In our example the decision was made inside the theorem for FALLIBLE, when the goal (HUMAN \$?X) was matched to the assertion (HUMAN TURING). PLANNER will retrace its steps, try to find a different assertion which matches the goal, find (HUMAN SOCRATES), and continue with the proof. The theorem will succeed with the value (FALLIBLE SOCRATES), and the THPROG will proceed to the next expression, (THGOAL (GREEK \$?X)). Since X has been assigned to SOCRATES, this will set up the goal (GREEK SOCRATES) which will succeed immediately by finding the

corresponding assertion in the data base. Since there are no more expressions in the THPROG, it will succeed, returning as its value the value of the last expression, (GREEK SOCRATES). The whole course of the deduction process depends on the failure mechanism for backing up and trying things over (this is actually the process of trying different branches down the subgoal tree.) All of the functions like THCOND, THAND, THOR, etc. are controlled by success vs. failure. Thus it is the PLANNER executive which establishes and manipulates subgoals in looking for a proof.

3.3.3 Differences with Other Theorem-Provers and Languages

Although PLANNER is written as a programming language, it differs in several critical ways from anything which is normally considered a programming language. First, it is goal-directed. Theorems can be thought of as subroutines, but they can be called through a very general pattern-matcher which looks at the goal which is to be satisfied. This is like having the ability to say "Call a subroutine which will achieve the desired result at this point." Second, the evaluator has the mechanism of success and failure to handle the exploration of the subgoal tree. Other languages, such as LISP, with a basic recursive evaluator have no way to do this. Third, PLANNER contains a bookkeeping system for matching patterns and manipulating a data base, and for handling that data base efficiently.

How is PLANNER different from a theorem prover? What is gained by writing theorems in the form of programs, and giving them power to call other programs which manipulate data? The key is in the form of the data the theorem-prover can accept. Most systems take declarative information, as in predicate calculus. This is in the form of expressions which represent "facts" about the world. These are manipulated by the theorem-prover according to some fixed uniform process set by the system. PLANNER can make use of imperative information, telling it how to go about proving a subgoal, or to make use of an assertion. This produces what is called hierarchical control structure. That is, any theorem can indicate what the theorem prover is supposed to do as it continues the proof. It has the full power of a general programming language to evaluate functions which can depend on both the data base and the subgoal tree, and to use its results to control the further proof by making assertions, deciding what theorems are to be used, and specifying a sequence of steps to be followed.

What does this mean in practical terms? In what way does it make a "better" theorem prover? We will give several

examples of areas where the approach is important.

First, consider the basic problem of deciding what subgoals to try in attempting to satisfy a goal. Very often, knowledge of the subject matter will tell us that certain methods are very likely to succeed, others may be useful if certain other conditions are present, while others may be possibly valuable, but not likely. We would like to have the ability to use heuristic programs to determine these facts and direct the theorem prover accordingly. It should be able to direct the search for goals and solutions in the best way possible, and able to bring as much intelligence as possible to bear on the decision. In PLANNER this is done by adding to our THGOAL statement a recommendation list which can specify that ONLY certain theorems are to be tried, or that certain ones are to be tried FIRST in a specified order. Since theorems are programs, subroutines of any type can be called to help make this decision before establishing a new THGOAL. Each theorem has a name (in our definition at the beginning of Section 3.1.1, the theorem was given the name THEOREM1), to facilitate referring to it explicitly.

The simplest kind of recommendation is THUSE, which takes a list of theorems (by names) and recommends that they be tried in the order listed. A more general recommendation uses filters which look at the theorem and decide whether it should be tried. The user defines his own filters, except for the standard filter

THTRUE, which accepts any theorem.

The filter command for theorems is THTBF, so a recommendation list of the form:

```
((THUSE TH1 TH2)(THTBF TEST)(THUSE TH-DESPERATION))
```

would mean to first try the theorem named TH1, then TH2, then any theorem which passes the filter named TEST (which the user would define), then if all that fails, use the theorem named TH-DESPERATION. In our programs, we have made use of only the simple capabilities for choosing theorems -- we do not define filters other than THTRUE. However, there is also a capability for filtering assertions in a similar way, and we do use this, as explained in section 4.3.

3.3.4 Controlling the Data Base

An important problem is that of maintaining a data base with a reasonable amount of material. Consider the first example above. The statement that all humans are fallible, while unambiguous in a declarative sense is actually ambiguous in its imperative sense (i.e. the way it is to be used by the theorem prover). The first way is to simply use it whenever we are faced with the need to prove (FALLIBLE \$?X). Another way might be to watch for a statement of the form (HUMAN \$?X) to be asserted, and to immediately assert (FALLIBLE \$?X) as well. There is no abstract logical difference, but the impact on the data base is tremendous. The more conclusions we draw when information is asserted, the easier proofs will be, since they will not have to make the additional steps to deduce these consequences over and over again. However since we don't have infinite speed and size, it is clearly folly to think of deducing and asserting everything possible (or even everything interesting) about the data when it is entered. If we were working with totally abstract meaningless theorems and axioms (an assumption which would not be incompatible with many theorem-proving schemes), this would be an insoluble dilemma. But PLANNER is designed to work in the real world, where our knowledge is much more structured than a set of axioms and rules of inference. We may very well, when we assert (LIKES \$?X POETRY) want to deduce and assert (HUMAN \$?X), since in

deducing things about an object, it will very often be relevant whether that object is human, and we shouldn't need to deduce it each time. On the other hand, it would be silly to assert (HAS-PART \$?X SPLEEN), since there is a horde of facts equally important and equally limited in use. Part of the knowledge which PLANNER should have of a subject, then, is what facts are important, and when to draw consequences of an assertion. This is done by having theorems of an antecedent type:

```
(DEFPROP THEOREM2
  (THANTE (X Y) (LIKES $?X $?Y)
    (THASSERT (HUMAN $?X)))
  THEOREM)
```

This says that when we assert that X likes something, we should also assert (HUMAN \$?X). Of course, such theorems do not have to be so simple. A fully general PLANNER program can be activated by an THANTE theorem, doing an arbitrary (that is, the programmer has free choice) amount of deduction, assertion, etc. Knowledge of what we are doing in a particular problem may indicate that it is sometimes a good idea to do this kind of deduction, and other times not. As with the CONSEQUENT theorems, PLANNER has the full capacity when something is asserted, to evaluate the current state of the data and proof, and specifically decide which ANTECEDENT theorems should be called.

PLANNER therefore allows deductions to use all sorts of knowledge about the subject matter which go far beyond the set of axioms and basic deductive rules. PLANNER itself is subject-

Independent, but its power is that the deduction processs never needs to operate on such a level of ignorance. The programmer can put in as much heuristic knowledge as he wants to about the subject, just as a good teacher would help a class to understand a mathematical theory, rather than just telling them the axioms and then giving theorems to prove.

3.3.5 Events and States

Another advantage in representing knowledge in an imperative form is the use of a theorem prover in dealing with processes involving a sequence of events. Consider the case of a robot manipulating blocks on a table. It might have data of the form, "block1 is on block2," "block2 is behind block3", and "if x is on y and you put it on z, then x is on z, and is no longer on y unless y is the same as z". Many examples in papers on theorem provers are of this form (for example the classic "monkey and bananas" problem). The problem is that a declarative theorem prover cannot accept a statement like (ON B1 B2) at face value. It clearly is not an axiom of the system, since its validity will change as the process goes on. It must be put in a form (ON B1 B2 S0) where S0 is a symbol for an initial state of the world. See <Green 1969> for a discussion of such "state" problems.

The third statement might be expressed as:

```
(FORALL (X Y Z S)(AND (ON X Y (PUT X Y S))
                      (OR(EQUAL Y Z)
                          (NOT(ON X Z (PUT X Y S))))))
```

In this representation, PUT is a function whose value is the state which results from putting X on Y when the previous state was S. We run into a problem when we try to ask (CN Z W (PUT X Y S)) i.e. Is block Z on block W after we put X on Y? A human knows that if we haven't touched Z or W we could just ask (ON Z W S) but in general it may take a complex deduction to

decide whether we have actually moved them, and even if we haven't, it will take a whole chain of deductions (tracing back through the time sequence) to prove they haven't been moved. In PLANNER, where we specify a process directly, this whole type of problem can be handled in an intuitively more satisfactory way by using the primitive function THERASE.

Evaluating (THERASE (ON \$?X \$?Y)) removes the assertion (ON \$?X \$?Y) from the data base. If we think of theorem provers as working with a set of axioms, it seems strange to have a function whose purpose is to erase axioms. If instead we think of the data base as the "state of the world" and the operation of the prover as manipulating that state, it allows us to make great simplifications. Now we can simply assert (ON B1 B2) without any explicit mention of states. We can express the necessary theorem as:

```
(DEFPROP THEOREM3
  (THCONSE (X Y Z) (PUT $?X $?Y)
    (THGOAL (ON $?X $?Z))
    (THERASE (ON $?X $?Z))
    (THASSERT (ON $?X $?Y)))
  THEOREM)
```

This says that whenever we want to satisfy a goal of the form (PUT \$?X \$?Y), we should first find out what thing Z the thing X is sitting on, erase the fact that it is sitting on Z, and assert that it is sitting on Y. We could also do a number of other things, such as proving that it is indeed possible to put X on Y, or adding a list of specific instructions to a movement plan for an arm to actually execute the goal. In a more complex

case, other interactions might be involved. For example, if we are keeping assertions of the form (ABOVE $?X$ $?Y$) we would need to delete those assertions which became false when we erased (ON $?X$ $?Z$) and add those which became true when we added (ON $?X$ $?Y$). ANTECEDENT theorems would be called by the assertion (ON $?X$ $?Y$) to take care of that part, and a similar group called ERASING theorems can be called in an exactly analogous way when an assertion is erased, to derive consequences of the erasure. Again we emphasize that which of such theorems would be called is dependent on the way the data base is structured, and is determined by knowledge of the subject matter. In this example, we would have to decide whether it was worth adding all of the ABOVE relations to the data base, with the resultant need to check them whenever something is moved, or instead to omit them and take time to deduce them from the ON relation each time they are needed.

Thus in PLANNER, the changing state of the world can be mirrored in the changing state of the data base, avoiding any need to make explicit mention of states, with the requisite overhead of deductions. This is possible since the information is given in an imperative form, specifying theorems as a series of specific steps to be executed.

If we look back to the distinction between assertions and theorems made on the first page, it would seem that we have established that the base of assertions is the "current state of

the world", while the base of theorems is our permanent knowledge of how to deduce things from that state. This is not exactly true, and one of the most exciting possibilities in PLANNER is the capability for the program itself to create and modify the PLANNER functions which make up the theorem base. Rather than simply making assertions, a particular PLANNER function might be written to put together a new theorem or make changes to an existing theorem, in a way dependent on the data and current knowledge. It seems likely that meaningful "learning" involves this type of behavior rather than simply modifying parameters or adding more individual facts (assertions) to a declarative data base.

3.3.6 PLANNER Functions

There are a number of other PLANNER commands, designed to suit a range of problem-solving needs. They are described in detail in <Hewitt 1969, 1970> and we will describe only those which are of particular use in our question answering program and which we will want to refer to later.

We have already mentioned the basic functions and described how they operate. THGOAL looks for assertions in the data base, and calls theorems to achieve goals. THAND takes a list of PLANNER expressions and succeeds only if they all succeed in the order they are listed. THOR takes a similar list and tries the expressions in order, but succeeds as soon as one of them does. Remember that in case of a failure farther along in the deduction, THOR can take back its decision and continue on down the list. The other simple LISP functions PROG, COND, and NOT have their PLANNER analogs, THPROG, THCOND, and THNOT, which operate just as their LISP counterparts, except that they are controlled by the distinction between "failure" and "success" instead of the distinction between NIL and non-NIL. THPROG acts like THAND, failing if any one of its members fails.

One of the most useful PLANNER functions is THFIND, which is used to find all of the objects or assertions satisfying a given PLANNER condition. For example, if we want to find all of the red blocks, we can evaluate:

```
(THFIND ALL $?X (X)
      (THGOAL(BLOCK $?X))
      (THGOAL(COLOR $?X RED)))
```

The function THFIND takes four pieces of information. First, there is a parameter, telling it how many objects to look for. When we use ALL, it looks for as many as it can find, and succeeds if it finds any. If we use an integer, it succeeds as soon as it finds that many, without looking for more. If we want to be more complex, we can tell it three things: a. how many it needs to succeed; b. how many it needs to quit looking, and c. whether to succeed or fail if it reaches the upper limit set in b.

Thus if we want to find exactly 3 objects, we can use a parameter of (3 4 NIL), which means "Don't succeed unless there are three, look for a fourth, but if you find it, fail".

The second bit of information tells it what we want in the list it returns. For our purposes, this will always be the variable name of the object we are interested in. The third item is a list of variables to be used, and the fourth is the body of the THFIND statement. It is this body that must be satisfied for each object found. It is identical to the body of a THPROG, and can have tags and THGO statements as well as a series of expressions to be evaluated.

Another function used extensively by the semantic interpreter is THAMONG. This takes two arguments, the second is a list, and the first is the name of a variable. If the

variable is assigned, THAMONG acts just like LISP MEMQ, succeeding if the value of the variable is contained in the list. However, if the variable is unassigned, THAMONG assigns it to the first member of the list, then succeeds. If this causes a failure to back up to the THAMONG, it binds the variable instead to the second member and tries again. This continues until the entire expression succeeds with some assignment or the list is exhausted, in which case THAMONG returns failure. Using this, along with the normal binding mechanism in THGOAL statements, failure can be used to run a loop through a list of objects which are specified by giving a PLANNER goal or arbitrary expression which they satisfy.

3.4 The BLOCKS World

We need a subject to discuss with our language-understanding program which gives a variety of things to say and in which we can carry on a discourse, containing statements, questions, and commands. We have chosen to pretend we are talking to a very simple type of robot (like the ones being developed in AI projects at Stanford and MIT) with only one arm and an eye. It can look at a scene containing toy objects like blocks and balls, and can manipulate them with its hand.

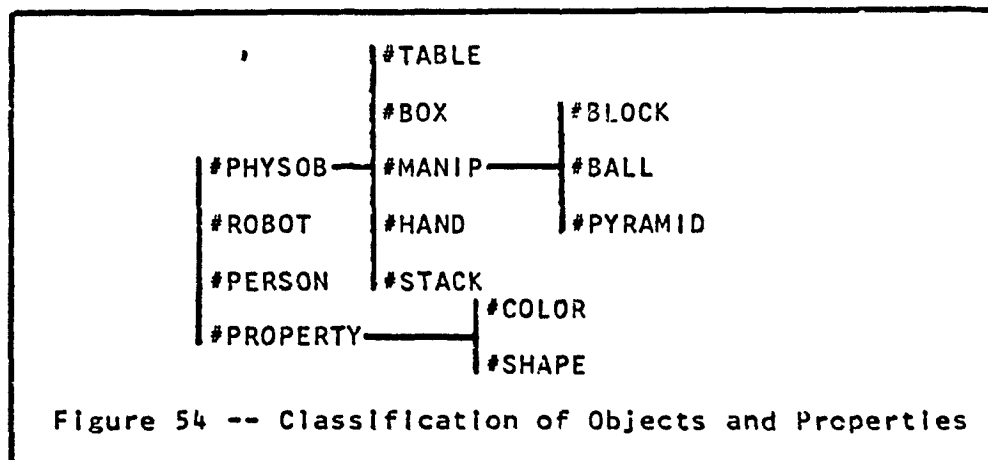
We have not tried to use an actual robot or to simulate it in physical detail. Since we are interested primarily in complex language activity, we have adopted a very simplified model of the world, and the "robot" exists only as a display on the CRT scope attached to the computer.

3.4.1 Objects

First we must decide what objects we will have in the world. In 3.1, we adopted some conventions for notation in representing objects and assertions. Any symbol which begins with ":" represents a specific object, while anything beginning with "#" is the name of a property or relation.

The model begins with the two participants in the discussion, the robot (named :SHRDLU), and the person (called :FRIEND). The robot has a hand (:HAND), and manipulates objects on a table (:TABLE), which has on it a box (:BOX). The rest of the physical objects are toys -- blocks, pyramids, and balls. We give them the names :B1, :B2, :B3,...

Next we must decide on the set of concepts we will use to describe these objects and their properties. We can represent these in the form of a tree:



The symbol #PHYSOB stands for "physical object", and #MANIP for "manipulable object" (i.e. something the robot can pick up).

We could use these as simple predicates, and have assertions like (#ROBOT :SHRDLU), (#HAND :HAND), and (#PYRAMID :B5) to say that Shrdlu is a robot, the hand is a hand, and :B5 is a pyramid. In section 4.4.3, we describe the way the language programs choose an English phrase to describe an object. In order to do so, they need a basic noun -- the one we would use to say "this is a ...". If we represented the concepts in the above tree using simple predicates, and then used the same form for other predicates, such as colors (for example, (#BLUE :B5)), the language generating routines would have no easy way to know which was the "basic" property. It would be necessary to keep lists and continually check. Instead, we adopt a different way of writing these concepts. We use the concept #IS to mean "has as its basic description", and write (#IS :SHRDLU #ROBOT), (#IS :HAND #HAND), and (#IS :B5 #PYRAMID).

Looking at the tree, we see that the properties #PHYSOB and #MANIP cannot be represented in this fashion, since any object having them also has a basic description. We therefore write (#MANIP :B5) and (#PHYSOB :TABLE).

Next, we would like to assign physical properties to these objects, such as size, shape, color, and location. Shape and color are handled with simple assertions like (#COLOR :BOX #WHITE) and (#SHAPE :B5 #POINTED). The possible shapes are #ROUND, #POINTED, AND #RECTANGULAR, and the colors are #BLACK,

#RED, #WHITE, #GREEN, and #BLUE. Of course it would involve no programming to introduce other shape or color names -- all that we would do is use them in an assertion, like (#COLOR :B11 #MAUVE), and add an assertion telling what type of thing they are. The property names themselves can be seen as objects, and we have the concepts #COLOR and #SHAPE, to make assertions like (#IS #BLUE #COLOR), and (#IS #RECTANGULAR #SHAPE).

Size and location are more complex, as they depend on the way we choose to represent physical space. We have adopted a standard three-dimensional coordinate system, with coordinates ranging from 0 to 1200 in all three directions. (The number 1200 was chosen for convenience in programming the display). The coordinate point (0 0 0) is in the front lower left-hand corner of the scene.

We have made the simplifying assumption that objects are not allowed to rotate, and therefore always keep their orientation aligned with the coordinate axes. We can represent the position of an object by giving the coordinates of its front lower left-hand corner, and can specify its size by giving the three dimensions. We use the symbols #SIZE and #AT, and put the coordinate triples as a single element in the assertions. For example, we might have (#AT :B5 (400 600 200)), and (#SIZE :B5 (100 100 300)).

Since we assume that the robot has an eye, the system begins the dialog with complete information about the objects in

the scene, their shapes, sizes, colors, and locations. In addition to the PLANNER assertions, the system keeps a table of sizes and locations for more efficient calculation when looking for an empty space to set something down.

3.4.2 Relations

The basic relations we will need for this model are the spatial relations between objects. Since we are interested in moving objects around in the scene, one of the most important relations is #SUPPORT. The initial data base contains all of the applicable support relations for the initial scene, and every time an object is moved, an antecedent theorem removes the old assertion about what was supporting it, and puts in the correct new one. We have adopted a very simplified notion of support, in which an object is supported by whatever is directly below its center of gravity, at the level of its bottom face. Therefore, an object can support several others, but there is only one thing supporting it. Of course this is an extreme simplification since it does not recognize that a simple bridge is supported. If this program were to be adapted to use with an actual robot, a much more general idea of support would be necessary. Along with the #SUPPORT relations, we keep track of the property #CLEARTOP. The assertion (#CLEARTOP X) will be in the data base if and only if there is no assertion (#SUPPORT X Y) for any object Y. It is also kept current by antecedent theorems which are called whenever an object is moved. This happens automatically whenever an assertion of the form (#AT OBJ (X Y Z)) is made. The theorems make the appropriate check to see whether the #CLEARTOP status of any object has changed, and if so the necessary erasures and assertions are made.

A second relation which is kept in the data base is #CONTAIN. The first participant must be the box, since this is the only container in the scene. The information about what is contained in the box is also kept current by an antecedent theorem. The relation #GRASPING is used to indicate what object (if any) the robot's hand is grasping. It is theoretically a two-place predicate, relating a grasper and a graspee, as in (#GRASPING :SHRDLU :B2). Since there is only one hand in our scene, it is clear who must be doing the grasping, so the assertion is reduced to (#GRASPING :B2).

The other relation which is stored in the data base is the #PART relation between an object and a stack. We can give a name to a stack, such as :S1, and assert (#PART :B2 :S1). As objects are moved, the changes to the data base are again made automatically by antecedent theorems which notice changes of location.

As we explained in section 3.3.3, we must decide what relations are useful enough to occupy space in our data base, and which should be recomputed from simpler information each time we need them. We have included relations like #SUPPORT and #CONTAIN because they are often referenced in deciding how to move objects. We can think of other relations, such as the relative position of two objects, which can be computed from their locations, and are not used often enough to be worth keeping in the data base and partially recomputing every time

something is moved. We represent these relations using the symbols #RIGHT, #BEHIND, and #ABOVE. (These represent the direction of the positive coordinate axis for X, Y, and Z respectively). We do not need the converse relations, since we can represent a fact like ":B1 is below :B2" by (#ABOVE :B2 :B1), and our semantic system can convert what is said to this standard format. The symbol #ON is used to represent the transitive closure of #SUPPORT. That is, Z is #ON A if A supports B, B supports C, ... supports Z.

The three spatial relations use a common consequent theorem called TC-LOC which decides if they are true by looking at the coordinates and sizes of the objects. The #ON relation has a consequent theorem TC-ON which looks for chains of support. (Notice that the prefix TC- stands for Theorem Consequent, and is attached to all of our consequent theorems. Similarly, TA- and TE- are used for antecedent and erasing theorems.)

The measurements of #HEIGHT, #WIDTH, and #LENGTH are represented as a simple assertion, like (#HEIGHT :B3 100), but they are not stored in the data base. They are computed when needed from the #SIZE assertion, and can be accessed by using the theorem TC-MEASURE, or by using a functional notation. The expression (#HEIGHT X) evaluates to the height of whatever object the variable X is bound to. If #SIZE is used in this way, it returns a measure of "overall size" to be used for comparisons like "bigger". Currently it returns the sum of the

X, Y, and Z coordinates, but it could be easily changed to be more in accord with human psychology.

In order to compare measurements, we have the relation #MORE. The sentence ":B1 is shorter than :B2" is equivalent to the assertion (#MORE #HEIGHT :B2 :B1). Again, we do not need the relation "less" since we can simply reverse the order of the objects. The relation #ASMUCH is used in the same way, to express "greater than or equal", instead of "strictly greater than". None of these assertions are stored (if we have ten objects, there will be almost 400 relationships), but are computed from more basic information as they are needed.

One final relationship is #LIKE, which relates a person or robot to any object. There is a theorem which shows that the robot likes everything, but knowledge about what the human user likes is gathered from his statements. The semantic programs can use statements about liking to generate further PLANNER theorems which are used to answer questions about what :FRIEND likes.

3.4.3 Actions

The only events that can take place in our world are actions taken by the robot in moving its hand and manipulating objects. At the most basic level, there are only three actions which can occur -- MOVETO, GRASP, and UNGRASP. These are the actual commands sent to the display routines, and could theoretically be sent directly to a physical robot system.

The result of calling a consequent theorem to achieve a goal requiring motion, like (#PUTON :B3 :B4), is a plan -- a list of instructions using the three elementary functions. MOVETO moves the hand and whatever it is currently grasping to a set of specified coordinates. GRASP sets an indicator that the grasped object is to be moved along with the hand, and UNGRASP unsets it. The robot grasps by moving its hand directly over the center of the object on its top surface, and turning on a "magnet". It can do this to any manipulable object, but can only grasp one thing at a time. Using these elementary actions, we can build a hierarchy of actions, including goals which may involve a whole sequence of deductions and actions, like #STACKUP.

The semantic programs never need to worry about details involving physical coordinates or specific motion instructions, but can produce input for higher-level theorems which do the detailed work.

At a slightly higher level, we have the PLANNER concepts

#MOVEHAND, #GRASP and #UNGRASP, and corresponding consequent theorems to achieve them. There is a significant difference between these and the functions listed above. Calling the function MOVETO actually causes the hand to move. On the other hand, when PLANNER evaluates a statement like:

```
(THGOAL(#MOVEHAND (600 200 300))(THUSE TC-MOVEHAND))
```

nothing is actually moved. The theorem TC-MOVEHAND is called, and it creates a plan to do the motion, but if this move causes us to be unable to achieve a goal at some later point, the PLANNER backup mechanism will automatically erase it from the plan. The robot plans the entire action before actually moving anything, trying all of the means it has to achieve its goal.

The theorems also do some checking to see if we are trying to do something impossible. For example, TC-MOVEHAND makes sure the action would not involve placing a block where there is already an object, and TC-UNGRASP fails unless there is something supporting the object it wants to let go of.

3.4.4 Carrying Out Commands

Some theorems, like TC-GRASP, are more complex, as they can cause a series of actions. In this section we will follow PLANNER through such an action, using the simplified theorems of figure 55. If PLANNER tries the goal:

```
(THGOAL (#GRASP :B1)(THUSE TC-GRASP))
```

the theorem TC-GRASP can do a number of things. It checks to make sure :B1 is a graspable object by looking in the data base for (#MANIP :B1). If the hand is already grasping the object, it has nothing more to do. If not, it must first get the hand to the object. This may involve complications -- the hand may already be holding something, or there may be objects sitting on top of the one it wants to grasp. In the first case, it must get rid of whatever is in the hand, using the the command #GET-RID-OF. The easiest way to get rid of something is to set it on the table, so TC-GET-RID-OF creates the goal (#PUTON \$?X :TABLE), where the variable \$?X is bound to the object the hand is holding. TC-PUTON must in turn find a big enough empty place to set down its burden, using the command #FINDSPACE, which performs the necessary calculations, using information about the sizes and locations of all the objects. TC-PUTON then creates a goal using #PUT, which calculates where the hand must be moved to get the object into the desired place, then calls #MOVEHAND to actually plan the move. If we look at the logical structure of our active goals at this point, assuming that we want to

```

(DEF THEOREM TC-CLEARTOP
  (THCONSE (X Y) (#CLEARTOP $X)
    GO (THCOND ((THGOAL (#SUPPORT $X $Y))
      (THGOAL (#GET-RID-OF $Y)
        (THUSE TC-GET-RID-OF))
      (THGO GO))
      ((THASSERT (#CLEARTOP $X))))))

(DEF THEOREM TC-GET-RID-OF
  (THCONSE (X Y) (#GET-RID-OF $X)
    (THOR
      (THGOAL (#PUTON $X :TABLE)(THUSE TC-PUTON))
      (THGOAL (#PUTON $X $Y)(THUSE TC-PUTON))))))

(DEF THEOREM TC-GRASP
  (THCONSE (X Y) (#GRASP $X)
    (THGOAL (#MANIP $X))
    (THCOND ((THGOAL (#GRASPING $X))
      ((THGOAL (#GRASPING $Y))
        (THGOAL (#GET-RID-OF $Y)
          (THUSE TC-GET-RID-OF))))
      (T))
    (THGOAL (#CLEARTOP $X) (THUSE TC-CLEARTOP))
    (THSETQ $Y (TOPCENTER $X))
    (THGOAL (#MOVEHAND $Y)
      (THUSE TC-MOVEHAND))
    (THASSERT (#GRASPING $X)))

(DEF THEOREM TC-PUT
  (THCONSE (X Y Z) (#PUT $X $Y)
    (CLEAR $Y (SIZE $X) $X)
    (SUPPORT $Y (SIZE $X) $X)
    (THGOAL (#GRASP $X) (THUSE TC-GRASP))
    (THSETQ $Z (TCENT $Y (SIZE $X)))
    (THGOAL (#MOVEHAND $Z) (THUSE TC-MOVEHAND))
    (THGOAL (#UNGRASP) (THUSE TC-UNGRASP)))

(DEF THEOREM TC-PUTON
  (THCONSE (X Y Z) (#PUTON $X $Y)
    (NOT (EQ $X $Y))
    (THGOAL (#FINDSPACE $Y $E (SIZE $X) $X $Z)
      (THUSE TC-FINDSPACE TC-MAKESPACE))
    (THGOAL (#PUT $X $Z) (THUSE TC-PUT)))

```

Figure 55 -- Simplified PLANNER Theorems

grasp :B1, but were already grasping :B2, we see:

```
(#GRASP :B1)
  (#GET-RID-OF :B2)
    (#PUTON :B2 :TABLE)
      (#PUT :B2 (453 201 0))
        (#MOVEHAND (553 301 100))
```

After moving, TC-PUT calls #UNGRASP, and we have achieved the first part of our original goal -- emptying the hand. Now we must clear off the block we want to grasp. TC-GRASP sets up the goal:

```
(THGOAL(#CLEARTOP :B2)(THUSE TC-CLEARTOP))
```

This is a good example of the double use of PLANNER goals to both search the data base and carry out actions. If the assertion (#CLEARTOP :B1) is present, it satisfies this goal immediately without calling the theorem. However if :B1 is not already clear, this THGOAL statement calls TC-CLEARTOP which takes the necessary actions.

TC-CLEARTOP will try to #GET-RID-OF the objects on top of :B1. This will in turn use #PUTON, which uses #PUT. But TC-PUT may have more to do this time, since the hand is not already grasping the object it has to move. It therefore sets up a goal to #GRASP the object, calling TC-GRASP. We have gone full circle, and are back in the theorem we started in. Of course this is no problem in a recursive language, and the process continues as it should until all of the goals have been achieved, or all of the methods included in the theorems have failed.

We have gone through this example in some detail to give a feeling for the goal-oriented programs used by PLANNER. The programs are highly recursive, with a clear subgoal structure, and with theorems often calling themselves to achieve subgoals. During all of this, PLANNER is keeping track of what is being done in such a way that it can back up and try something different if necessary. For example, if TC-GET-RID-OF puts an object on the table, and it later is in the way of something which must be done, a failure will propagate back to TC-GET-RID-OF, and it will try putting it somewhere else.

Figure 56 lists the different action concepts more systematically, showing the form of the goal statements, and the actions taken by the theorems corresponding to them. In our program, the names of the theorems are formed by adding the prefix TC- for a consequent theorem, TA- for antecedent, and TE- for erasing. This is strictly for programmer convenience in recognizing the purpose of a theorem from its name. All of these goals can be used internally within the BLOCKS system, and most of them can be called directly by the semantic programs, as direct translations of English commands. Some, like #MOVEHAND cannot be called by the linguistic programs, since the semantic routines do not include a way to specify exact coordinates in English.

Command	Effect
(#MOVEHAND (X Y Z))	Move the center of the hand to location (X Y Z). Anything being grasped goes along automatically.
(#UNGRASP)	Let go of whatever the hand is holding. Fails if the object is not supported.
(#GRASP X)	Grasp object X, doing any manipulations necessary to get to it.
(#PUT W (X Y Z))	Put W at location (X Y Z). Fails unless the space is empty.
(#RAISEHAND)	Raise the hand and whatever it holds as high as it will go.
(#PICKUP X)	Grasp X and raise the hand.
(#PUTON X Y)	Put object X on object Y. If there is not enough free space on Y, move objects to make it.
(#PUTIN X Y)	The same as #PUTON, except that Y must be a box, and different methods are used to find space.
(#GET-RID-OF X)	Try to put X on the table, and if that fails put it on anything else.
(#CLEARTOP X)	Get rid of everything which is on X.
(#STACKUP (X Y...))	Stack X, Y, ... on top of each other, in order of size.
(#FINDSPACE A (X Y Z) B \$_C)	This goal can be achieved by two different theorems. TC-FINDSPACE tries to find a space of size (X Y Z) on top of object A, counting any space occupied by B as empty. \$_C is a variable binding used to return the answer. If this fails, TC-MAKESPACE can create the space by moving objects.

Figure 56 -- BLOCKS Commands

3.4.5 Memory

In order to answer questions about past events, the BLOCKS programs remember selected parts of their subgoal tree. They do this by creating objects called events, and putting them on an EVENTLIST. The system does not remember the detailed series of specific steps like #MOVEHAND, but keeps track of the larger goals like #PUTON and #STACKUP. The time of events is measured by a clock which starts at 0 and is incremented by 1 every time any motion occurs. The theorems which want to be remembered use the functions MEMORY and MEMOREND, calling MEMORY when the theorem is entered and MEMOREND when it exits. MEMOREND causes an event to be created, combining the original goal statement with an arbitrary name (chosen from E1, E2,...). Recall from Section 3.1 that a relation can include its own name, so that other relations can refer to it. If we call TC-PUTON with the goal (#PUTON \$?X \$?Y), with the variables X and Y bound to :B1 and :B2 respectively, the resulting event which is put into the data base is (#PUTON E1 :B1 :B2). The event name is second, instead of last as described in 3.1 for unimportant technical reasons which will be changed in later versions.

In addition to putting this assertion in the data base, MEMOREND puts information on the property list of the event name -- the starting time, ending time, and reason for each event. The reason is the name of the event nearest up in the subgoal tree which is being remembered. The reason for goals called by

the linguistic part of the system is a special symbol meaning "because you asked me to". MEMORY is called at the beginning of a theorem to establish the start time and declare that theorem as the "reason" for the subgoals it calls.

A second kind of memory keeps track of the actual physical motions of objects, noting each time one is moved, and recording its name and the location it went to. This list can be used to establish where any object was at any past time.

When we want to pick up block :B1, we can say:
(THGOAL(#PICKUP :B1)), and it is interpreted as a command. How can we ask "Did you pick up :B1?"? When the robot picked it up, an assertion like (#PICKUP E2 :B1) was stored in the data base. Therefore if we ask PLANNER

```
(THPROG(X)
  (THGOAL (#PICKUP $?X :B1)))
```

it will find the assertion, binding the variable X to the event name E2. Since the property list of E2 gives its starting and ending times, and its reason, this is sufficient information to answer most questions.

If we want to ask something like "Did you pick up :B1 before you built the stack?" we need some way to look for particular time intervals. This is done by using a modified version of the event description, including a time indicator. The exact form of the time indicator is described in the section on semantics, but the way it is used to establish a goal is:

```
(THGOAL(#PICKUP $?X :B1 $?TIME)(THUSE TCOTE-PICKUP))
```

The prefix TCTE- on the name of a theorem means that it includes a time and an event name. Ordinarily when such a theorem is entered, the variable TIME would have a value, while the variable X would not. The theorem looks through the data base for stored events of the form (#PICKUP \$?X :B1) and checks them to see if they agree with the time TIME.

For some events, like #PUTON, this is sufficient since the system remembers every #PUTON it does. For others, like #PICKUP less information is kept. When #PICKUP is called as a goal at the top level, it is remembered. But the system does not remember each time something was picked up in the course of moving the toys around. The fact that a block was picked up can be deduced from the fact that it was put somewhere, and the theorem TCTE-PICKUP actually looks at a number of different types of events (like #PUTON and #PUTIN) to find all the occasions on which an object was really picked up.

For spatial relations, we also need to be able to include time, for example, "Was the block behind the pyramid before...?" In this case, no assertions are stored, since the memory of motion events is sufficient to reconstruct the scene. There are special theorems with the prefix TCT- which try to verify a relation with a time condition. For example, we can ask "Is :B1 on :B2?" with the goal

```
(THGOAL(#ON :B1 :B2)(THUSE TC-ON))
```

To ask "Was :B1 on :B2 before..." we bind the variable TIME to

the representation of the time we are interested in, and ask

```
(THGOAL(#ON :B1 :B2 $?TIME)(THUS& TCT-ON))
```

The theorem TCT-ON is the same as TC-ON except that it deals with the specified time instead of the present. Similar TCT-theorems exist for all of the spatial relations, and for properties which change in time, such as #CLEARTOP and #AT.

Appendix C is a listing of parts of the BLOCKS program as it was used in generating the sample dialog. PLANNER theorems are sufficiently self-explanatory that only a few comments have been included. The auxiliary LISP functions (such as the one which decides whether a space is empty) are not included, but are available on request, along with the complete BLOCKS listings.

Chapter 4. Semantics

4.1 What Is Semantics?

4.1.1 the Province of Semantics

The field of semantics has always been a hazy swampland. There is little agreement among "semanticists" where its borders lie or what the terrain looks like. Logicians, philosophers, and linguists all approach it with the tools of their own trade, and the problem of just defining "semantics" and "meaning" have occupied volumes of debate.

In trying to program computers to understand natural language, it has been necessary to have a more explicit and complete notion of semantics. The attempts at writing language understanding programs have made it more clear just what a semantic theory has to do, and how it must connect with the syntactic and logical aspects of language. In practical terms, we need a transducer which can work with a syntactic analysis, and produce data which is acceptable to a logical deductive system.

In the preceding chapters we have described the two ends of a language system -- a syntactic parser with a grammar of English, and a deductive system with a base of knowledge about a particular subject. What does our semantic theory have to do to

fill the gap?

In section 3.1.2 we outlined the basis for a theory of semantics. It includes a world of "concepts" and structures of concepts which are postulated by the linguist in trying to explain linguistic phenomena. These are not a psychological reality, but a formalism in which he can systematically express those aspects of meaning which are relevant to language use. By manipulating structures in this formalism as a part of analyzing sentences in natural language, the theory can directly deal with problems of relating meaning to parts of the speaker's and hearer's knowledge which are not mentioned explicitly in the sentence being analyzed.

A semantic theory must describe the relationship between the words and syntactic structures of natural language and the postulated formalism of concepts and operations on concepts. In our theory, this relationship is described as a set of procedures which analyze linguistic forms to produce representations of meaning in the internal conceptual formalism. Just as with the grammar, this does not purport to be a model of an actual process taking place in the hearer or speaker. The process description is used because it is a powerful way to describe "neutral" relationships, as well as being psychologically suggestive.

The theory must describe relationships at three different levels. First, there must be a way to define the meanings of words. We pointed out in the section on "meaning" (section 3.1) that the real "meaning" of a word or concept cannot be defined in simple dictionary terms, but involves its relationship to an entire vocabulary and structure of concepts. However, we can talk about the formal description attached to a

word which allows it to be integrated into the system. In the rest of this chapter, we will use the word "meaning" in this more limited sense, describing those formal aspects of the meaning of a word (or syntactic construction) which are attached to it as its dictionary definition.

The formalism for definitions should not depend on the details of the semantic programs, but should allow users to add to the vocabulary in a simple and natural way. It should also be possible to handle the quirks and idiosyncracies of meaning which words can have, instead of limiting ourselves to "well-behaved" standard words.

At the next level we must relate the meanings of the words in a sentence to each other and to the meaning of the syntactic structures. We need an analysis of the ways in which English structures are designed to convey meaning, and what role the different words and syntactic features play in this meaning.

Finally, a sentence in natural language is never interpreted in isolation. It is always part of a context, and its meaning is dependent on that context. A theory should explain the different ways in which the "setting" of a sentence can affect its meaning. It must deal both with the linguistic setting (the context within the discourse) and the real-world setting (the way meaning interacts with knowledge of non-linguistic facts.)

4.1.2 The Semantic System

With definite goals in mind for a semantic system, we can consider how to implement it. First let us look at what it should know about English. As we have been emphasizing throughout the paper, a language is not a set of abstract symbols. It is a system for conveying meaning, and has evolved with very special mechanisms for conveying just those aspects of meaning needed for human communication.

Section 3.1 discussed the person's "model of the world" which is organized around notions of "objects", having "properties" and entering into "relationships." In 3.1.3, these are combined to form more complicated logical expressions. Looking at the properties of English syntax (as described in Section 2.3) we see that these basic elements of the "world model" are just what English is good at conveying.

For describing objects, there is the NOUN GROUP. It contains a noun, which indicates the kind of object; adjectives and classifiers, which describe further properties of the object; and a complex system of quantifiers and determiners describing its logical status -- whether it is a particular object, ("the sun"), a class of objects ("people"), a particular set of objects ("John's lizards"), an unspecified set containing a specified number of objects ("three bananas"), etc. The details (described in section 4.2) are complex, but the important thing is the existence of a systematic structure.

For describing relationships and events, there are the CLAUSE, PREPOSITION GROUP, and ADJECTIVE GROUP. The CLAUSE is especially suited for dealing with relationships having a particular time reference, working in coordination with the VERB GROUP, which functions to convey information about time, using an ingenious system of tenses. Clauses can also be used to represent an event or relationship as an object (as in "His going pleased me."), or to modify a particular object within a NOUN GROUP (in "the man who broke the bank"). The PREPG is a less flexible and simpler way of expressing relationships which do not need modifiers such as time, place, and manner (such as "the man in the blue vest"). The ADJG is used in some constructions to describe properties and some special kinds of relationships of objects (such as "Her gift was bigger than a breadbox.")

The semantic system is built around a group of about a dozen programs which are experts at looking at these particular syntactic structures. They look at both the structures and the meanings of the words to build up PLANNER expressions which will be used by the deductive mechanism. It is important to remember that the parser uses systemic grammar so the semantic programs can look directly for features such as PASSIVE or PLURAL or QUESTION to make decisions about the meaning of the sentence or phrase.

Since each of these semantic "specialists" can work

separately, there is no need to wait for a complete parsing before beginning semantic analysis. The NOUN GROUP specialist can be called as soon as a NOUN GROUP has been parsed, to see whether it makes sense before the parser goes on. In fact, the task can be broken up, and a preliminary NOUN GROUP specialist can be called in the middle of parsing (for example, after finding the noun and adjectives, but before looking for modifying clauses or prepositional phrases) to see whether it is worth continuing, or whether the supposed combination of adjectives and noun is nonsensical. The grammar is in the form of a program, so it is just as easy to call a semantic routine at any time as a syntactic one. Any semantic program has full power to use the deductive system, and can even call the grammar to do a special bit of parsing before going on with the semantic analysis. For this reason it is very hard to classify the semantic analysis as "top-down" or "bottom-up". In general each piece of the structure is analyzed as it is parsed, which is a bottom-up approach. However whenever there is a reason to delay a part of the analysis until some of the larger structure has been analyzed, it is just as easy to write the semantic specialist programs in this top-down manner. In our system both approaches are used.

4.1.3 Words

A semantic system needs to deal with two different kinds of words. Some words are included in the general knowledge of the English language. Words like "that" or "than", in "He knew that they were madder than hornets." would be difficult to define except in terms of their place in the sentence structure. They are being used as signals of certain syntactic structures and features, and have no meaning except for this signalling (which is recognized by the grammar). These are often called "function words" in distinction to the "content words" which make up the bulk of our vocabulary. This is not a sharp distinction, since many words serve a combination of purposes (for example, numbers are basically "function words", but each one has its unique meaning). We can generally distinguish between words like "that" and "than" whose meanings are built into the system, and words like "snake", "under", and "walk", which surely are not.

The definitions of content words should not have to include "expert" knowledge about the semantics or grammar of the language. In defining the word "mighty", we should not have to worry about whether it appears in "The sword is mighty," or "the mightiest warrior", or "a man mightier than a locomotive." We should be able to say "'Mighty' means having the property represented conceptually as #MIGHT.", and let the semantic system do the rest.

We need a semantic language for expressing definitions in a

way which does not depend on the grammar or the particular semantic programs. Each of our "specialists" which looks at the meanings of words should be able to interpret those statements in the semantic language which might be relevant to its job.

Section 4.2 describes simple formats for defining common verbs, nouns, adjectives, and prepositions, and in fact, these definitions do not look much like programs at all. Why then do we call this a "language" instead of saying that we have a set of special formats for defining words? The distinction becomes important for all of the irregular cases and the idiosyncracies that words can have. For example, in "The block is on the roof of the car.", "the roof of the car" is a NG referring to a particular object which is a roof. But if we say "The block is on the right of the box", we are not referring to a particular object which is a "right". The normal NG mechanism for describing objects is being used instead to describe a relationship between the block and the box. We could reprogram our NOUN GROUP semantic specialist to recognize this special case and treat it differently, but this is a path leading to a roadblock. We will not be able to anticipate every case, and as the program becomes more and more patched, it will become harder to change and less likely to work.

What we need is a flexible way of defining words, so that the word "right" itself can cause the right things to happen in semantic interpretation, without changing the system. This is

achieved by letting the definition of each word be a LISP program to be run at an appropriate time in the semantic analysis. For simple cases, there are standard functions with a special format for usual types of definitions. In the complex cases there is a platform from which to operate, doing whatever calculations and changes of the environment are needed. This flexibility is important in many places. For example, the word "one" when used as a noun (as in "the green one") has a special use for referring back to previously mentioned nouns. It could not be defined by a simple format, as could "block" or "dog", since it involves complex decisions about what is really being referred to, and needs access to the previous discourse. In our system, its definition as a noun is compatible with the definitions of all other nouns -- the semantic specialists don't know anything about it. When the NG specialist is ready to use the definition of the noun, it calls it as a program. In the usual case, this program sets up a standard data structure. In the case of "one", it calls a heuristic program for understanding back-references, and its effect on the meaning will depend on the discourse. Similarly, the verb "be" is called like any other verb by the semantic specialist, but in fact its definition is a complex program describing its different uses.

The use of procedures to represent meanings of words gives a flexibility which allows these exceptional words to be handled

as well as the more ordinary forms. At the same time, it provides a strict test of representations of procedures for particular words, since the procedures can actually be run in the system.

4.1.4 Ambiguity

A semantic theory must have some way to account for multiple meanings of words, phrases, and sentences. We would like to explain not only how multiple interpretations can occur, but also how the hearer sorts them out to pick a single meaning.

As a start, we must allow words to have several "senses", and must be able to have multiple interpretations of phrases and sentences to correspond to them. Next we must realize that the syntactic structures can also lead to semantic ambiguities. Sentences like the famous "Time flies like an arrow." derive some of their ambiguity from their ability to be analyzed syntactically in more than one way. Finally, we include some ambiguities as a result of the semantic analysis. The sentence "A man sitting in this room fired the fatal shot." will be ambiguous even if we agree on a single meaning for each word, and a surface structure for the sentence. If spoken by Perry Mason at a dramatic moment in the courtroom, it means "a man who is sitting in this room", but if spoken by the detectives when they broke into the empty hotel room across the street from the scene of the crime, it means "who was sitting in this room". This could be treated as a syntactic ambiguity in the deep structure, but in our analysis it is instead treated as a semantic ambiguity involving the time reference.

In describing the grammar it was pointed out that we do not carry forward simultaneous parsings of a sentence. We try to

find the "best" parsing, and try other paths only if we run into trouble. In semantics we take the other approach. If a word has two meanings, then two semantic descriptions are built simultaneously, and used to form two separate phrase interpretations.

We can immediately see a problem here. There is dire danger of a combinatorial explosion. If words A, B, C, and D each have three meanings, then a sentence containing all of them may have $3 \times 3 \times 3 \times 3$, or 81 interpretations. The possibilities for a long sentence are astronomical.

Of course a person does not build up such a tremendous list. As he hears a sentence, he "filters out" all but the most reasonable interpretations. We know that a "ball" can be either a spherical toy or a dancing party, and that "green" can mean either the color green, or unripe, or inexperienced. But when we see "the green ball", we do not get befuddled with six interpretations, we know that only one makes sense. The use of "green" for "unripe" applies only to fruit, the use as "inexperienced" applies only to people, and the color only to physical objects. The meaning of "ball" as a party fits none of these categories, and the meaning as a "spherical toy" fits only the last one. We can subdivide the world into rough classes such as "animate", "inanimate", "physical", "abstract", "event", "human", etc. and can use this classification scheme to filter out meaningless combinations of interpretations.

Some semantic theories <Fodor> are based almost completely on this idea. We would like to use it for what it is -- not a complete representation of meaning, but a rough classification which eliminates fruitless semantic interpretations. Our system has the ability to use these "semantic markers" to cut down the number of semantic interpretations of any phrase or sentence.

A second method used to reduce the number of different semantic interpretations is to do the interpretation continuously. We do not pile up all possible interpretations of each piece of the sentence, then try to make logical sense of them together at the end. As each phrase is completed, it is understood. If we come across a phrase like "the colorful ball" in context, we do not keep the two different possible interpretations in mind until the utterance is finished. We immediately look in our memory to see which interpretation is meaningful in the current context of discourse, and use only that meaning in the larger semantic analysis of the sentence. Since our system allows the grammar, semantics and deduction to be easily intermixed, it is possible to do this kind of continuous interpretation.

Finally we must deal with cases where we cannot eliminate all but one meaning as "senseless". There will be sentences where more than one meaning makes sense, and there must be some way to choose the correct one in a given context. In the section on context below, we discuss the use of the overall

discourse context in assigning a plausibility factor to a particular interpretation. By combining the plausibilities of the various parts of a sentence, we can derive an overall factor to help choose the best.

There will always be cases where no set of heuristics will be enough. There will be multiple interpretations whose plausibilities will be so close that it would be simply guessing to choose one. In our sample dialogue, there is an example with the word "on". "The block is on top of the pyramid." could mean either "directly on the surface" or "somewhere above". There is no way for the hearer (or computer) to read minds. The obvious alternative is to ask the speaker to explain more clearly what is meant. As a final resort, the system can ask questions like "By the word 'on' in the phrase 'on top of green blocks' did you mean 'directly on the surface' or 'somewhere above'?". The methods used for handling ambiguity are described in more detail in section 4.2.10

4.1.5 Discourse

At the beginning of our discussion of semantics, we discussed why a semantic system should deal with the effect of "setting" on the meaning of a sentence. A semantic theory can account for three different types of context.

First, there is the local discourse context, which covers the discourse immediately preceding the sentence, and is important to semantic mechanisms like pronoun reference. If we ask the question "Did you put it on a green one?" or "Why?" or "How many of them were there then?", we assume that it will be possible to fill in the missing information from the immediate discourse. There are a number of special mechanisms for using this kind of information, and they form part of a semantic theory.

Second, there is an overall discourse context. A hearer will interpret the sentence "The group didn't have an identity." differently depending on whether he is discussing mathematics or sociology. There must be a systematic way to account for this effect of general subject matter on understanding. In addition to the effects of general subject on choosing between meanings of a word, there is an effect of the context of particular things being discussed. If we are talking about Argentina, and say "The government is corrupt.", then it is clear that we mean "the government of Argentina". If we say "Pick up the pyramid.", and there are three pyramids on the table, it will

not be clear which one is meant. But if this immediately follows the statement "There is a block and a pyramid in the box.", then the reference is to the pyramid in the box. This would have been clear even if there had been several sentences between these two. Therefore this is a different problem than the local discourse of pronoun reference. A semantic theory must deal with all of these different forms of overall discourse context.

Finally, there is a context of knowledge about the world, and the way that knowledge effects our understanding of language. If we say "The city councilmen refused the demonstrators a permit because they feared violence.", the pronoun "they" will have a different interpretation than if we said "The city councilmen refused the demonstrators a permit because they advocated revolution." We understand this because of our sophisticated knowledge of councilmen, demonstrators, and politics -- no set of syntactic or semantic rules could interpret this pronoun reference without using knowledge of the world. Of course a semantic theory does not include a theory of political power groups, but it must explain the ways in which this kind of knowledge can interact with linguistic knowledge in interpreting a sentence.

Knowledge of the world may affect not only such things as the interpretation of pronouns, but may alter the parsing of the syntactic structures as well. If we see the sentence "He hit

the car with a rock." the structure will be parsed differently from "He hit the car with a dented fender.", since we know that cars have fenders, but not rocks.

In our system, most of this discourse knowledge is called on by the semantic specialists, and by particular words such as "one", "it", "then", "there", etc. We have concentrated particularly on local discourse context, and the ways in which English carries information from one sentence to the next. A number of special pieces of information are kept, such as the time, place, and objects mentioned in the previous sentence. This information is referenced by special structures and words like pronouns, "then", and "there". The meaning of the entire previous sentence can be referred to in order to answer a question like "Why did you do that?" or just "Why?".

There are two facilities for handling overall discourse context. The first is a mechanism for assigning a "plausability factor" to an interpretation of a word. For example, the definition of the word "bank" might include the fact that if we are discussing money, it is most likely to mean a financial institution, while if we are discussing rivers it probably means the edge of the land. Our system allows the definition of a word to include a program to compute a "plausability factor" (an arbitrary additive constant) for each interpretation. This computation might involve looking at the rest of the sentence for key words, or might use some more

general idea, like keeping track of the general area of discussion (perhaps in some sort of network or block structure) and letting the plausibility of a particular meaning depend on its "distance" from the current topic. This has not been implemented since we have included only a single topic of discourse in the vocabulary. It is discussed further in section 5.2.

The second type of overall discourse context involves the objects which have been previously mentioned. Whenever an object or one of its properties is mentioned, either by the human or the computer, a note is made of the time. Later, if we use a phrase like "the pyramid", and the meaning is not clear, the system can look for the one most recently mentioned.

Finally, the knowledge of the world can enter into the semantic interpretation. We have mentioned that the grammar can ask the semantic interpreter "Does this NOUN GROUP make sense?" before continuing the parsing. The semantics programs can in turn call on PLANNER to make any deductions needed to decide on its sensibility. Thus information about the world can guide the parsing directly.

4.1.6 Goals of a Semantic Theory

We have set ourselves very broad goals in our definition of semantics, asking for everything which needs to be done, rather than limiting ourselves to those aspects which can be explained and characterized in a neat formalism. How does this compare with the more limited goals of a semantic theory like that of Fodor and Katz (<Fodor>), which looks only at those aspects of meaning which are independent of the "setting" of a sentence?

We have seen that their theory of "semantic markers" is in fact a part of the "filtering" needed for "exploiting semantic relations in the sentence to eliminate potential ambiguities" (<Fodor>p. 485)", and that the "semantic distinguishers" are a rudimentary form of the logical descriptions which we build up to describe objects and events. They state that "the distinction between markers and distinguishers is meant to coincide with the distinction between that part of the meaning of a lexical item which is systematic for the language and that part of the meaning of the item which is not." (<Fodor> p. 498). We believe that much more of meaning is systematic, and that a semantic theory can be of a much wider scope.

What about the more restricted goals a semantic theory might achieve such as "accounting for... the number and content of the readings of a sentence, detecting semantic anomalies, and deciding upon paraphrase relations between sentences."? In a more complete semantic theory, these are not primary goals, but by-products of the analysis. A phrase is a semantic anomaly if the system produces no possible interpretations for it. Two sentences are paraphrases if they produce the same representation in the internal formalism for meaning, and the "number and content" of the readings of a sentence are the

immediate result of its semantic analysis. Which of these will happen depends on the entire range of ways in which language communicates meaning, not on a restricted subset such as the logical relations of markers. Once we have a conceptual representation for meaning, problems such as these are secondary byproducts of the basic analysis which relates a sentence to the representation of its meaning.

In addition, we can talk about sentences being anomalies or paraphrases "in context", as well as "without regard to context", since we want the theory to include a systematic analysis of those features of context which are relevant to understanding.

4.2 Semantic Structures

The previous section outlined the structure of a semantic interpreter, and described the use of semantic "specialists" in analyzing different aspects of linguistic structure. Each specialist is in the form of a procedure which describes the particular aspect of semantics which with it is concerned. We can look at its function as creating a part of a complete description of the meaning of the sentence by building complex list structures which we will call "semantic structures" to describe objects and relationships. Events are a type of relationship (involving time), and the class of "object" includes anything which could be treated as an object in English grammar, even if it is as abstract as "truth". There are two basic types of structures used -- one to describe objects, (an Object Semantic Structure, or OSS) and the other to describe relationships (an RSS). In general, noun groups are interpreted to form object structures, while the other groups and clauses are interpreted to form relationship structures. Words already have a semantic structure of their own (their definition) and are used in building up the structures for the larger units which contain them.

4.2.1 Object Semantic Structures

Let us first look at the semantic structures used to describe objects. First, we need the actual PLANNER statements which will be used in deducing things about the objects. An NG like "a red cube" can be described using the formalism of Chapter 3:

```
(THPROG (X1)
  (THGOAL(#IS $X1 #BLOCK))
  (#EQDIM $X1)
  (THGOAL(#COLOR $X1 #RED)))
```

Figure 57 -- Simple PLANNER Description

The variable "X1" represents the object, and this description says that it should be a block, it should have equal dimensions, and it should be red. (See section 3.4 for the details of representation). A phrase such as "a red cube which supports three pyramids but is not contained in a box" has a more complex description. This would be built up from the descriptions for the various objects, and would end up:

```
(THPROG(X1)
  (THGOAL(#IS $X1 #BLOCK))
  (#EQDIM $X1)
  (THGOAL(#COLOR $X1 #RED))
  (THFIND 3 $X2 (X2) (THGOAL(#IS $X2 #PYRAMID))
    (THGOAL(#SUPPORT $X1 $X2)))
  (THNOT(THPROG(X3)
    (THGOAL(#IS $X3 #BOX))
    (THGOAL(#CONTAIN $X3 $X1))))))
```

Figure 58 -- PLANNER Description

We can learn how the semantic specialists work by watching them build the pieces of this structure. First take the simpler

NG, "a red cube". The first NG specialist doesn't start work until after the noun has been parsed. The PLANNER description is then built backwards, starting with the noun, and continuing in right-to-left order through the classifiers and adjectives. The beginning of the NG, with the determiner, number, and ordinal is handled by a part of the NG specialist described later. The first NG specialist is named SMNG1 -- all of the names begin with SM (for "semantic"), followed by the name of the unit they work with, followed by a number indicating the order in which they are called. SMNG1 sets up an environment (we will describe various parts of it as we go), then calls the definition of the noun. (Remember that definitions are in the form of programs). For simple nouns there is a standard function to define them easily. What should the definition include? First, a way to indicate the PLANNER statements which are the heart of its meaning. The symbol "***" is used to represent the object, so our definition of "cube" contains the expression:

```
((#IS *** #BLOCK)(#EQDIM ***))
```

The syntax of PLANNER functions such as THPROG and THGOAL will be added by the specialists, since we want to keep the definition as simple as possible.

There is one other part of the definition for a noun -- the semantic markers, used to filter out meaningless interpretations of a phrase. The definition needs to attach

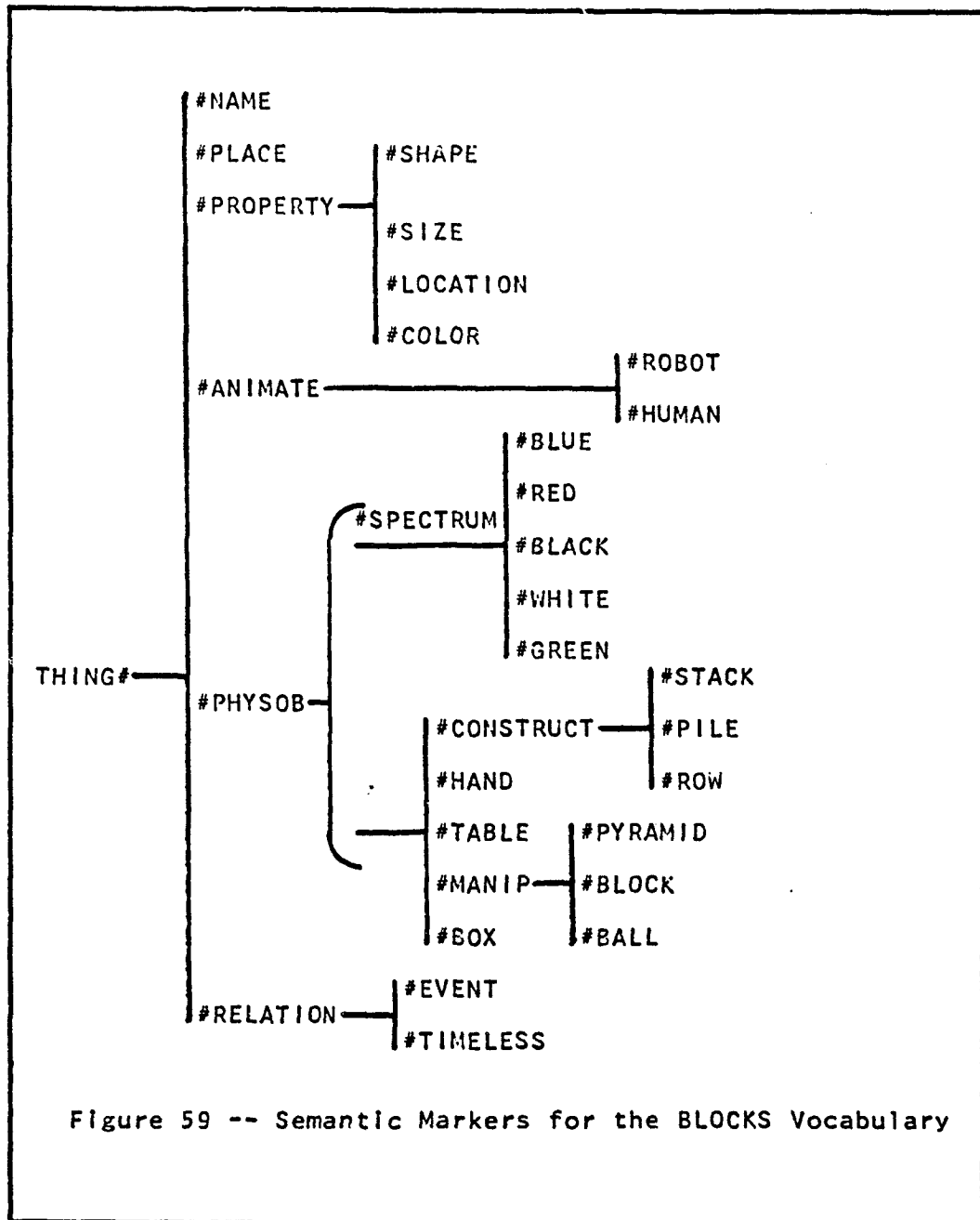
these semantic markers to each OSS. The BLOCKS world uses the tree of semantic markers in Figure 59.

This is the same type of diagram used for grammars, in which vertical bars represent choices of mutually exclusive markers, while horizontal lines represent logical dependency. The symbol "#PHYSOB" means "physical object", and "#MANIP" means "manipulable object". The word "cube" refers to an object with the markers (#THING #PHYSOB #MANIP #BLOCK). We shouldn't need to mention all of these in the definition, since the presence of #BLOCK implies the others through the logical structure of the marker tree.

The definition of the noun "cube" is then:

```
(NMEANS((#BLOCK)((#IS *** #BLOCK)(#EQDIM ***))))
```

NMEANS is the name of the function for dealing with nouns, and it accepts a list of different meanings for a word. In this case, there is only one meaning. The first part of the definition is the marker list, followed by the reduced PLANNER definition. When NMEANS is executed, it puts this information onto the semantic structure which is being built for the object. It takes care of finding out what markers are implied by the tree, and deciding which predicates need to be in a THGOAL statement (like #IS), and which are LISP predicates (like #EQDIM). We will see later how it also can decide what recommendation lists to put onto the PLANNER goals, to guide the deduction.



SMNG1 then calls the definition for the adjective "red". We would like this definition to include the PLANNER assertion (#COLOR *** #RED), and indicate that it applies only to physical objects. We can use the same format used for nouns, defining "red" as:

```
(NMEANS((#PHYSOB)((#COLOR *** #RED))))
```

Notice that there is no distinction made between the use of #PHYSOB here to imply "applies only to physical objects" and the use of #BLOCK in the definition of "cube" to say "this is a block". This is because of the way the markers are implemented. The marker list in a definition is interpreted to mean "this definition applies only if none of the markers here are in conflict with any of the markers already established for the object". Since the noun is the first thing interpreted, its markers cannot possibly conflict, and are simply entered as the initial marker list for the object. The marker programs are designed so that we do not need to limit ourselves to a single tree -- we could classify objects along several dimensions, and set up separate marker trees for each. For example, we might classify objects both by their physical properties and by their use.

The order of analysis of modifiers is quite natural to the use of "relative" modifiers. It is impossible to give an absolute definition for "big" or "little", since a "big flea" is still not much competition for a "little elephant". The meaning

of the adjective is relative to the noun it modifies. In fact, it may also be relative to the adjectives following it as well. A "big toy elephant" is on a scale of its own. Since our system analyzes the NG from right to left, the meaning of each adjective is added to the description already built up for the head and modifiers to the right. Since each definition is a program, it can just as well be a program which examines the description (both the semantic markers and the PLANNER description), and produces an appropriate meaning relative to the object being described. This might be in the form of an absolute measurement (e.g. a "big elephant" is more than 12 feet tall) or can remain in a relative form by producing a PLANNER expression of the form "the number of objects fitting the description and smaller than the one being described is more than the number of suitable objects bigger than it is".

In adding the meaning of "red" to the semantic structure, the specialist must make a choice in ordering the PLANNER expressions. We remember from section 2.3 that the order of expressions can be important, since variable assignments are done in the order encountered. If we have the first sequence shown in Figure 60, PLANNER will look through all of the blocks, checking until it finds one which is red. However if we have the second, it will look through all of the red objects until it finds one which is a block. In the robot's tiny world, this isn't of much importance, but if we had a data base which could

take phrases like "a man in this room", we would certainly be better off looking around the room first to see what was a man, than looking through all the men in the world to see if one was in the room.

```
(THPROG(X)
  (THGOAL(#IS $X #BLOCK))
  (THGOAL(#COLOR $X #RED)))

(THPROG(X)
  (THGOAL(#COLOR $X #RED))
  (THGOAL(#IS $X #BLOCK)))
```

Figure 60 -- Ordering Goals

To make this choice we allow each predicate (like #IS or #COLOR) to have associated with it a program which knows how to evaluate its "priority" in any given environment. The program might be as simple as a single number, which would mean "this relation always has this priority". It might on the other hand be a complex heuristic program which takes into account the current state of the world and the discussion. In our definitions, we have adopted the simpler alternative, assigning fixed priorities in the range 0 to 1000 arbitrarily. By keeping track of the priority of the expression currently at the top of the PLANNER description, the function NMEANS can decide whether to add a new expression above or below it.

Let us now look at the actual structure which would be built up by the program:

((((X1) 200 (THGOAL(#IS \$?X1 #BLOCK))	PLANNER
(THGOAL(#COLOR \$?X1 #RED))	description
(#EQDIM \$?X1))	
(0 #BLOCK #MANIP #PHYSOB #THING)	markers
(#MANIP #PHYSOB #THING)	systems
X1	variable
(NS INDEF NIL)	determiner
NIL)	ordinal

Figure 61 -- OSS for "a red cube"

Most of the parts of this structure (called an Object Semantic Structure or OSS) have already been explained. The PLANNER description includes a variable list (we will see its use later), the priority of the first expression, and a list of PLANNER expressions describing the object. The "markers" position lists all of the semantic markers applicable to the object. The 0 at the beginning of the list is the "plausability" of this interpretation. This factor was discussed in section 4.1.4, and is set when we are faced with more than one possible interpretation of a word. Each semantic structure carries along with it an accumulated plausability rating. This will remain 0 unless it is set specifically by an ambiguity.

The "systems" position is a list of all of the nodes in the set of marker trees (remember that there can be more than one) which have already had a branch selected. It is used in looking for marker conflicts. The "variable" is the variable

name chosen to represent this object. The system generates it from the set X_1, X_2, X_3, \dots , providing a new one for each new structure. The only two positions left are the determiner and the ordinal. These are explained in section 4.2.4

4.2.2 Relative Clauses

Let us now take a slightly more complicated NG, "a red cube which supports a pyramid," and follow the parsing and semantic analysis. First, the NG parsing program finds the determiner ("a"), adjective ("red"), and noun ("cube"). At this point SMNG1 is called and creates the structure described in the previous section. Notice that the NG is not finished when SMNG1 is called -- It has only reached a point where we can do a first analysis. At this point, the NG might be rejected without further parsing if the combination of noun, classifiers, and adjectives is contradictory to the system of semantic markers.

Next the NG program looks for a qualifier, and calls the CLAUSE part of the grammar by (PARSE CLAUSE RSQ). The feature RSQ (rank shifted qualifier) informs the CLAUSE program that it should look for a RELWD like "which". It does, and then looks for a VG, succeeding with "supports". The VG program calls its own semantic specialist to analyze the time reference of the clause, but we will ignore this for now. Next, since "support" is transitive, the CLAUSE looks for an object, and calls the NG program. This operates in the same way as before, producing a semantic structure to describe "a pyramid". The definition of "pyramid" is:

```
(NMEANS((#PYRAMID)((#IS *** #PYRAMID))))
```

so the resulting structure is:

```

( (X2) 200 (THGOAL(#IS $?X2 #PYRAMID)))
  (0 #PYRAMID #MANIP #PHYSOB #THING)
  (#MANIP #PHYSOB #THING)
X2
(NS INDEF NIL)
NIL)

```

Figure 62 -- OSS for "a pyramid"

At this point the first CLAUSE specialist is called to analyze the clause "which supports a pyramid". We want to define verbs in a simple way, as we do nouns and adjectives, saying something like "If the subject and object are both physical objects, then "support" means the relation #SUPPORT between them in that order". This is written formally using the function CMEANS, as:

```
(CMEANS((((#PHYSOB)))(#PHYSOB)))(#SUPPORT #1 #2)NIL))
```

All of the extra parentheses are there to leave room for fancier options which will be described later. The important parts are the semantic marker lists for the objects participating in the relationship, and the actual PLANNER expression naming it. The symbols "#1" and "#2" (and "#3" if necessary) are used to indicate the objects, and the normal order is 1. semantic subject (SMSUB) 2. semantic first object (SMOB1) 3. semantic second object (SMOB2). Notice that we have prefixed the word "semantic" to each of these. In fact, they may very well not be the actual syntactic subject and objects of the clause. In this example, the SMSUB is the NG "a red cube" to which the clause is being related. SMCL1 knows

this since the parser has noted the feature SUBJREL. Before calling the definition of the verb, SMCL1 has found the OSS describing "a red cube" and set it as the value of the variable SMSUB. Similarly it has taken the OSS for "a pyramid" and put it in SMOB1, since it is the object of the clause. The definition of the verb "support" is now called, and CMEANS uses the information in the definition to build up a Relation Semantic Structure (RSS). First it checks to make sure that both objects are compatible with their respective marker lists. The marker lists are in the same order as the symbols #1, #2, and #3. In this case, both the subject and object must be physical objects.

Next SMCL1 substitutes the objects into the relation. If it inserted the actual semantic structures, the result would be hard to read and time-consuming to print. Instead, the NG specialists assign a name to each OSS, from the set NG1, NG2, NG3,... We therefore get (#SUPPORT NG1 NG2) as the description of the relationship. The final semantic structure for the clause (after a second specialist, SMCL2 has had a chance to look for modifiers and rearrange the structure into a convenient form) is:

(NG1	(#SUPPORT NG1 NG2)	NIL)	(0))
rel	relation	neg	markers

Figure 63 -- Relation Semantic Structure 1

The position marked "rel" holds the name of the NG description to which this clause serves as a modifier. We will see later that it can be used in a more general way as well. The "relation" is the material for PLANNER to use, and "neg" marks whether the clause is negative or not.

The last element is a set of semantic markers and a priority, just as we had with object descriptions. Relationships have the full capability to use semantic markers just as objects do, and at an early stage of building a relation structure, it contains a PLANNER description, markers, and systems in the identical form to those for object structures (this is to share some of the programs, such as those which check for conflicts between markers). We can classify different types of events and relationships (for example those which are changeable, those which involve physical motion, etc.) and use the markers to help filter out interpretations of clause modifiers. For example, the modifying PREPG "without the shopping list" in "He left the house without the shopping list" has a different interpretation from "without a hammer" in "He built the house without a hammer." If we had a classification of activities which included those involving motion and those using

tools, we could choose the correct interpretation. A system can be constructed which operates much like Fillmore's case system, assigning classes of verbs according to the type of modification they take, and using this to find the correct relation between a verb and its modifying phrase. This will be discussed more in the section on types of PREPG.

In our limited world, we have not set up a marker tree for relationships and events, so we have not included any markers in the definition of "support". The marker list in the RSS therefore contains only the plausibility, 0. The "NIL" in the definition indicates that there are no markers, and would be replaced by a list of markers if they were used.

The clause is now finished, and the specialist on relative clauses (SMRSQ) is called. Its task is to take the information contained in the PLANNER descriptions of the objects involved in the relation, along with the relation itself, and to put it all onto the PLANNER description of the object to which the clause is being related. The way in which this is done depends on the exact form of the different objects (particularly on their determiners). In this case, it is relatively easy, and the description of "a red cube which supports a pyramid" becomes:

```

( ( ((X1 X2) 200 (THGOAL(#IS $?X1 #BLOCK))
      (THGOAL(#COLOR $?X1 #RED))
      (#EQDIM $?X1)
      (THGOAL(#IS $?X2 #PYRAMID))
      (THGOAL(#SUPPORT $?X1 $?X2)))

  (0 #BLOCK #MANIP #PHYSOB #THING)
  (#MANIP #PHYSOB #THING))
X1
(NS INDEF NIL)
NIL)

```

Figure 64 -- OSS for "a block which supports a pyramid"

The only thing which has changed is the PLANNER description, which now holds all of the necessary information. Its variable list contains both X1 and X2, and these variable names have been substituted for the symbols NG1 and NG2 in the relation, which has been combined with the separate PLANNER descriptions for the objects. Section 4.2.4 describes how a relative clause works with other types of NG descriptions.

4.2.3 Preposition Groups

Comparing the phrase "a red cube which supports a pyramid" with the phrase "a red cube under a pyramid, we see that relative clauses and qualifying prepositional phrases are very similar in structure and meaning. In fact, their semantic analysis is almost identical. The definition of a preposition like "under" uses the same function as the definition of a verb like "support", saying "if the semantic subject and object are both physical objects, then the object is #ABOVE the subject" (Remember, that in our BLOCKS world we chose to represent all vertical space relations using the concept #ABOVE). This can be formalized as:

```
(CMEANS((((#PHYSOB)))(#PHYSOB)))(#ABOVE #2 #1)NIL)
```

Again, the symbols #1 and #2 refer to the semantic subject and semantic first object, but in the case of a preposition group used as a qualifier, the SMSUB is the NG of which the PREPG is a part, while the SMOBJ is the object of the PREPG (the PREPOBJ). As with clauses, the situation may be more complex. For example, in a sentence like "Who was the antelope I saw you with last night?", the SMOBJ of the PREP "with" is the question element "who" in the MAJOR CLAUSE. However, the PREPG specialist (SMPREP) takes care of all this, and in defining a preposition, we can deal directly with the SMSUB and the SMOBJ. Notice that if we had been defining "above" instead of "under", everything would have been the same except that the relation

would have been (#ABOVE #1 #2) instead of (#ABOVE #2 #1). If the PREPG is an adjunct to a CLAUSE, the SMSUBJ is the RSS defining the CLAUSE. The definition of a preposition can then use the semantic markers which are included in an RSS.

4.2.4 Types of Object Descriptions

In the examples so far, all of the objects described have been singular and INDEFinite, like "a red cube", and the semantic system has been able to assign them a PLANNER variable and use it in building their properties into the description. Let us consider another simple case, a DEFINite object, as in "a red cube which supports the pyramid".

The analysis begins exactly as it did for the earlier case, building a description of "red cube", then one of "pyramid." The "pyramid" description differs from OSS 2 in having DEF in place of INDEF in its determiner. This is noted at the very beginning of the analysis, but has no effect until the entire NG (including any qualifiers) has been parsed. At that time, the second NG specialist SMNG2 checks for a definite NG and tries to determine what it refers to before going on (we have pointed out in various places how this is used to guide the parsing). It takes the PLANNER description which has been built up, and hands it to PLANNER in a THFIND ALL expression. The result is a list of all objects fitting the description. Presumably if the speaker used "the", he must be referring to a particular object he expects the listener to be aware of. If more than one object fits the description, there are various discourse heuristics used to find the reference, (see Section 4.3.3) and if nothing succeeds, a failure message is produced and the parser has to back up and try something else to parse the NG.

If SMNG2 is able to find the object being referred to, it puts it into the description (on the property list). When SMRSQ relates the descriptions to build the meaning of "a red cube which supports the pyramid" it takes advantage of this. The object found will have a proper name like :B5. Instead of building the PLANNER description of OSS 3, it builds:

```
((X1) 200 (THGOAL(#IS $?X1 #BLOCK))
          (#EQDIM $?X1)
          (THGOAL(#SUPPORT $?X1 :B5)))
```

Figure 65 -- PLANNER Description 1
"a red cube which supports the pyramid"

The object itself is used in the relation rather than dealing with its description.

What if we had asked about "a red cube which supports three pyramids"? In that case the PLANNER description would include an expression using the function THFIND with a numerical parameter, as shown in Figure 66. If we had said "a red cube which supports at most two pyramids", a fancier THFIND parameter would have been used, as shown. Here, the parameter means "be satisfied if you don't find any, but if you find 3, immediately cause a failure." In addition to numbers, the SMNG1 and RSQ programs can work together to relate descriptions of quantified objects. "A red cube which supports some pyramid" is handled just like the original indefinite case. "A red cube which supports no pyramid" and "a red cube which supports every pyramid" are handled using the other PLANNER primitives. A universal quantifier is translated as "there is no pyramid which

```
(THGOAL(#IS $?X2 #PYRAMID))
(THGOAL(#SUPPORT $?X1 $?X2))
```

"which supports a pyramid"

```
(THGOAL(#SUPPORT $?X1 :B3))
```

"which supports the pyramid"

```
(THFIND 3 $?X2 (X2) (THGOAL(#IS $?X2 #PYRAMID))
(THGOAL(#SUPPORT $?X1 $?X2)))
```

"which supports three pyramids"

```
(THFIND (0 3 NIL) $?X2 (X2) (THGOAL(#IS $?X2 #PYRAMID))
(THGOAL(#SUPPORT $?X1 $?X2)))
```

"which supports at most two pyramids"

```
(THNOT
(THPROG (X2) (THGOAL(#IS $?X2 #PYRAMID))
(THGOAL(#SUPPORT $?X1 $?X2))))
```

"which supports no pyramids"

```
(THNOT
(THPROG (X2) (THGOAL(#IS $?X2 #PYRAMID))
(THNOT
(THGOAL(#SUPPORT $?X1 $?X2)))))
```

"which supports every pyramid"

Figure 66 -- Quantifiers

the red cube does not support". For the robot, "every" means "every one I know about". This is not a requirement of PLANNER, or even of the way we have set up our semantic programs. It was done as a convenience, and will be changed when the system is expanded to discuss universal statements as well as the specific commands and questions it now handles.

We similarly handle the whole range of quantifiers and types of numbers, using the logical primitives and THFIND parameters of PLANNER. The work is actually done in two places. SMNG1 takes the words and syntactic features, and generates the "determiner" which was one of the ingredients of our semantic structure for objects. The determiner contains three parts. First, the number is either NS (singular, but not with the specific number "one"), NPL (plural with no specific number), NS-PL (ambiguous between the two, as in "the fish"), or a construction containing an actual arithmetic number. This can either be the number alone, or a combination with ">", "<", or "exactly". Thus the two NGs "at most two days" and "fewer than three days" produce the identical determiner, containing "< 3)". The second element of the determiner is either DEF, INDEF, ALL, NO, or NDET (no determiner at all -- as in "We like sheep.") The third is saved for the question types HOWMANY and WHICH, so it is NIL in a NG which is not a QUEST or REL.

Number	
NS	an apple
NPL	some thoughts
7	seven sisters
(> 2)	at least three ways
(< 5)	fewer than five people
(EXACTLY 2)	exactly two minutes
Determiner	
DEF	the law
INDEF	a riot
ALL	every child
NO	nothing
NDET	good intentions
Question Marker	
HOWMANY	how many years
WHICH	which road
Figure 67 -- Examples of Determiner Elements	

Other specialists such as SMRSQ and the answering routines use this information to produce PLANNER expressions like the ones described above. In addition, there are special programs for cases like the OF NG, as in "all of your dreams". In this case, the PREPOBJ following "of" is evaluated as a NG first. Therefore in "three of the blocks", we analyze "the blocks" first, and since it is definite, PLANNER is called to find out what it refers to. It returns a list of "the blocks", (e.g. (:B1 :B4 :B6 :B7)). The OF specialist uses the PLANNER function THAMONG (which chooses its variable bindings from "among" a given list) to produce an expression like:

```
(THFIND 3 $?X1 (X1) (THAMONG X1 (QUOTE(:B1 :B4 :B6 :B7))))
```

Ordinals are treated specially, along with SUPERlative ADJectives. If we have a NG like "the biggest block which supports a pyramid", it is impossible for SMNG1 to add the meaning of "biggest" to the description in the same way as it would add an expression for "big". The block is "biggest" with respect to a group of objects, and that group is not fully defined until the entire NG has been parsed, including the qualifiers. SMNG1 therefore does a partial analysis of the meaning, looking up the name of the measure that particular adjective refers to, then hangs the result in the last niche of the OSS described in section 4.2.1. After all has been parsed, SMNG2 finds it there and creates a full logical description. In the case of "the biggest block which supports a pyramid", we would get the PLANNER description:

```
((X1 X2 X3 X4 ) 200
  (THGOAL(#IS $?X1 #BLOCK))
  (THGOAL(#IS $?X2 #PYRAMID))
  (THGOAL(#SUPPORT $?X1 $?X2))
  (THNOT
    (THAND(THGOAL(#IS $?X3 #BLOCK))
      (THGOAL(#IS $?X4 #PYRAMID))
      (THGOAL(#SUPPORT $?X3 $?X4))
      (THGOAL(#MORE #SIZE $?X3 $?X1))))
```

Figure 68 -- PLANNER Description 2
"the biggest block which supports a pyramid"

A similar type of description is generated for other superlatives and ordinals.

4.2.5 The Meaning of Questions

So far, we have discussed the semantics of objects and the relationships which are used to describe them in preposition groups and relative clauses. Now we will deal with the overall meaning of a sentence as an utterance -- as a statement, a question, or a command. The sentence is analyzed into a relationship semantic structure, and the system must act on it by responding, taking an action, or storing some knowledge.

First let us look at questions. In describing the grammar of clauses (see section 2.3.3) we pointed out the similarities between questions and relative clauses, which share a large part of the system network and the parsing program. They also have much in common on a semantic level. We can look at most questions as being a relative clause to some focus element in the sentence.

In the class of WH questions, this resemblance is easy to see. First we can take a NGQ question, whose question element is a NG. The question "Which red cube supports a pyramid?" is very closely related to the NG "a red cube which supports a pyramid. The system can answer such a question by relating the clause to the object, and building a description of "a red cube which supports a pyramid." It then takes this entire PLANNER description and puts it into a THFIND ALL statement, which is evaluated in PLANNER. The result is a list of objects fitting the description, and is in fact the answer to our question. Of

course PLANNER might find several objects or no objects meeting the description. In this case we need answers like "none of them" or "two of them". Section 4.4 describes how responses to questions such as these are generated, depending on the relation between the specific question and the data found. If the question is "how many" instead of "which", the system goes through the identical process, but answers by counting rather than naming the objects found.

No matter what type of NGQ we have (there is a tremendous variety -- see section 2.3.3) the same method works. We treat the MAJOR clause as a relative clause to the NG which is the question element, and which we call the focus. This integrates the relationship intended by the clause into the description of that object. PLANNER then finds all objects satisfying the expanded description, and the results are used to generate an answer.

Next, we have the QADJ questions, like "when", "why", and "how". In these cases the focus is on an event rather than on one element of the relation. If we ask "Why did you pick up a block?", we are referring to an event which was stored in the system's memory as (#PICKUP E23 :B5) where :B5 is the name of the object picked up, and E23 is the arbitrary name which was assigned to the event (see Section 3.4 for a description of the way such information is stored.) We can ask in PLANNER:

```
(THFIND ALL $?EVENT ($?EVENT $?X)
  (THGOAL(#PICKUP $?EVENT $?X))
  (THGOAL(#IS $?X #BLOCK)))
```

In other words, "Find all of the events in which you picked up a block." This is clearly the first thing which must be done before we can answer "why". Once it has been done, answering is easy, since PLANNER will return as the value of THFIND a list of names of such events. On the property list of an event we find the name of the event for which it was called as a subgoal (the "reason"). We need only to describe this in English. Similarly if the question is "when", the property list of the event gives its starting and ending times. If the question is "how" it takes a little more work, since the subgoal tree is stored with only upward links. But by looking on the EVENTLIST, the system can generate a list of all those goals which had as their reason the one mentioned in the sentence.

This concept of a relation as a sort of object called an "event" is useful in other parts of the semantics as well -- for instance in dealing with embedded clauses as in "the block which I told you to pick up". This is described in section 4.2.12.

"Where" is sometimes handled differently, as it may be either a constituent of a clause, such as a location object (LOBJ) (in "Where did you put it?") or an ADJUNCT (as in "Where did you meet him?"). The first case is handled just like the NG case, making the clause a relative, as if it were "the place where you put it", then asking in PLANNER:

```
(THFIND ALL $?PLACE (PLACE EVENT)
  (THGOAL (#PUT $?EVENT :OBJ $?PLACE)))
```

The ADJUNCT case involves thinking about a special #LOCATION assertion, as in:

```
(THFIND ALL $?PLACE (PLACE EVENT)
  (THGOAL(#MEET $?EVENT :YOU :HIM))
  (THGOAL(#LOCATION $?EVENT $?PLACE)))
```

In this example, we have moved away from the BLOCKS world since it does not yet contain any actions in its vocabulary which occur at a specific place without that place being mentioned in the event, such as #PUT. However the semantic system is perfectly capable of handling such cases.

So far, we have seen that we can answer WH- questions by pretending they are a relative to some object, event, or place, and by adding the relationship to the description of this focus. It is an interesting fact about English that even in a YES-NO question, where there is no question element there is usually a focus. Consider a simple question like "Does the box contain a block?" Someone might answer "Yes, a red one.", as if the question had been "Which block does the box contain?" Notice that "Yes, the box." would not have been an appropriate answer. Something about "the box" makes it obvious that it is not the focus. It is not its place as subject or object, since "Is a block in the box?" reverses these roles, but demands the same answer. Clearly it is the fact that "a block" is an INDEFINITE NG.

The fact that a speaker says "a block" instead of "the

block" indicates that he is not sure of a specific object referred to by the description. Even if he does not inquire about it specifically, the listener knows that the information will be new, and possibly of interest since he mentioned the object. In answering "Does the box contain a block?", our system does the same thing it would do with "How many blocks does the box contain?". It adds the relation "contained by the box" to the description of "a block", and finds all of the objects meeting this description. Of course the verbal answer is different for the two types of question. In one case, "Yes" is sufficient, while in the other "one" is. But the logical deduction needed to derive it is identical. In fact, our system uses this extra information by replying, "Yes, two of them: a red one and a green one." This may sometimes be verbose, but in fact gives a natural sound to the question-answering. It takes on the "intelligent" character of telling the questioner information he would be interested in knowing, even when he doesn't ask for it explicitly.

In YES-NO questions, it is not always easy to determine the focus. Only an INDEF NG which is not embedded in another NG can be the focus, but there may be several of them in a sentence. Sometimes there is no way to choose, but that is rare. In asking a question, people are usually focusing their attention on a particular object or event. There are a number of devices for indicating the focus. For example a quantifier, like "any"

or a TPRON like "something" emphasizes the NG more than a simple determiner like "a". In both "Does anything green support a block?", and "Does a block support anything green?", the phrase "anything green" is the focus. When none of these cues are present, the syntactic function of the NG makes a difference. If we ask "Is there a block on a table", then "block" is the focus, since it is the subject while "table" is inside a PREPG. Our system contains a heuristic program which takes into account the kind of determiners, number features (singular is more likely than plural), syntactic position, and other such factors in choosing a focus. If it is in fact very difficult to choose in a given case, it is likely that the speaker will be satisfied with any choice.

For sentences in the past tense, which contain no focus NG, we can again have an event as a focus. If we ask, "Did Jesse James rob the stagecoach?", a possible answer, interpreting the event as the focus, is "Yes, three times: yesterday, last week, and a year ago." This is closely parallel to answering questions in which the focus is an object.

There are some questions which have no focus, such as present-tense clauses with only definite noun groups. These, however, are even easier to answer, since they can be expressed in the form of a simple set of assertions with no variables. The NG analysis finds the actual objects referred to by a definite NG, and these are used in place of the variable in

relationships. We can therefore answer "yes" or "no" by making a goal of the relationship and letting PLANNER evaluate it. The question "Does the red cube support the box?" would generate the simple PLANNER expression

```
(THGOAL (#SUPPORT :B3 :BOX))
```

if :B3 is the internal name for the red cube. PLANNER would return a non-NIL value only if the answer were "yes".

4.2.6 Interpreting Imperatives

The system can accept commands in the form of IMPERATIVE sentences. These are handled somewhat differently from questions. If they contain only definite objects, they can be treated in the way mentioned above for questions with no focus. The command "Pick up the red ball.", is translated into the relationship (#PICKUP :B7) which can be evaluated directly by putting it in a THGOAL statement which will carry out the action:

```
(THGOAL (#PICKUP :B7)(THUSE TC-PICKUP))
```

However, if we say "Pick up a red ball.", the situation is different. We could first use THFIND to find a red ball, then put this object in a simple goal statement as we did with "the red ball". This, however, might be a bad idea. In choosing a red ball arbitrarily, we may choose one which is out of reach or which is supporting a tower. The robot might fall or be forced to do a lot of work which it could have avoided with a little thought.

We want to send the theorem which works on the goal a description rather than an object name, and let the theorem choose the specific object to be used, according to the criteria which best suit it. This is the method we have adopted. Remember that each OSS has a name like "NG45". Before a clause is related to its objects, these are the symbols used in the relationship.

When we analyze "Pick up a red ball", it will actually produce (#PICKUP NG45), where NG45 names an OSS describing "a red ball." We use this directly as a goal statement, calling a special theorem which knows how to use these descriptions. The theorem calls a theorem named TC-FINDCHOOSE, which uses the description of the object, along with a set of "desirable properties" associated with objects used for trying to achieve the goal. #PICKUP may specify that it would prefer picking up something which doesn't support anything, or which is near the hand's current location. Each theorem can ask for whatever it wants. Of course, it may be impossible to find an object which fits all of the requirements, and the theorem has to be satisfied with what it can get. TC-FINDCHOOSE tries to meet the full specifications first, but if it can't find an object (or enough objects in the case of plural), it gradually removes the restrictions. It must always keep the full requirements of the description input in English in order to carry out the specified command. The robot simply tries to be clever about choosing those objects which fit the command but are also the easiest for it to use.

4.2.7 Accepting Declarative Information

In addition to questions and commands, the system can accept declarative sentences. We have intentionally not emphasized them, as there are theoretical problems and dangers in designing a program to accept information in this way. In Chapter 3, we discussed the complex world-model a person has and explained why we felt that intelligence needed a highly structured and coordinated body of knowledge rather than a set of separate uniform facts or axioms. It is comparatively easy to get a program to add new information of the second type, but very difficult to get it to add the first, since this involves understanding the relationship between the new information and whatever is already there.

Therefore, although we have included declarative sentences in our dialog (and they are fully handled in the grammar), we believe that before trying to "tell" many things to a program, we need to have a better idea of how knowledge should be structured, and the program should approach new information as a problem solving activity rather than a clerical one.

When a human sees a new sentence, he does not simply store it away, but he relates it to what he already knows, perhaps changing his "programs", or ignoring the content of the sentence and interpreting something about the person who said it. A language understander needs to have an interpreter which looks at each new sentence and decides how to use it. This may include checking it for consistency with what it already knows, creating new data or types of data in its storage, modifying theorems, and many other possibilities. This is discussed

further in Section 5.1

In our system we have four different ways in which information can be accepted in a declarative sentence. The first is a simple word definition facility. If we say "A 'marb' is a red block which is behind a box.", the system recognizes that we are defining a new word. It currently recognizes this by the quote marks, but it could just as easily declare all unfamiliar words as possible new words. We have not done this as it would eliminate the feature that the system immediately recognizes typing errors without waiting to begin parsing the sentence.

In this kind of definition, the complement of the sentence is a noun group, which has an OSS. We save this OSS and generate a new dictionary entry for the word, defined syntactically as a noun, and with its semantic definition being the program "set the object description to this one we saved earlier." Remember that all definitions are programs, so this one fits in with no problem. When it is called on to build part of the description, it simply inserts the description used to define it. If we talk about "two big marbs", the system will build a description exactly like the one for "two big red blocks which are behind a box."

The second kind of information the system accepts is simple assertions involving a predicate for which it does not have complete knowledge. As we mentioned in Section 3.4, the system

has complete data about the physical characteristics of the objects in the scene. We have selected #LIKE as an arbitrary relation about which the system knows nothing except what it is told in the dialog. If we say "I like you." this produces the assertion (#LIKE :FRIEND :SHRDLU) (the name of the robot is :SHRDLU) which is simply added to the data base. The system also plays a trick with the adjective "nice". Instead of having some concept of #NICE, it assumes that the use of "nice" in describing something is really saying more about the speaker than the object, so the definition of "nice" is

```
(NMEANS((#THING)((#LIKE :FRIEND ***))))
```

In other words, the person who uses the word "nice" likes the object he was referring to.

If we use an object which isn't definite, as in "I like red blocks.", the system uses the object description to generate a simple PLANNER consequent theorem. It creates a theorem of the form:

```
(THCONSE (X1)
  (#LIKE :FRIEND $?X1)
    (THGOAL (#IS $?X1 #BLOCK))
    (THGOAL (#COLOR $?X1 #RED)))
```

This theorem says "Whenever you want to prove that the user likes something, you can do it by proving that it is a block and it is red." This is added to the theorem data base, and can be used to answer questions or carry out deductions involving objects described as "nice". The system does not separate types of non-definite objects and assumes universal quantification.

The results would have been the same if the sentence used "any red block", "every red block", "all red blocks", or (wrongly) "a red block." A more complete treatment is one of the possible extensions of the system.

It does notice the form "no red blocks" and uses this for the fourth kind of information. It sets up an almost identical theorem, but with a "kicker" at the end. If we say "I like no red blocks.", it sets up the theorem:

```
(THCONSE (X1)
  (#LIKE :FRIEND $?X1)
    (THGOAL (#IS $?X1 #BLOCK))
    (THGOAL (#COLOR $?X1 #RED)))
  (THFAIL THGOAL))
```

When the system is trying to prove that we like something, this theorem is called just like the one above. But this time, after it finds out that the object is a red block, it does not succeed. Instead, it uses the PLANNER function THFAIL in a powerful way. Instead of just causing that theorem to fail, it causes the entire goal to fail, regardless of what other theorems there are. We can also accept a sentence like this with a positive NG but a negative clause, as in "I don't like the red block" or "I don't like any red blocks."

4.2.8 Time

One of the most complex parts of English semantics is the way of establishing temporal relationships. It was pointed out earlier that one of the primary differences between the clause and other units such as the NG or PREPG is the special set of mechanisms within the clause for handling time. In this section we will describe how those mechanisms operate both within the clause and at other levels of syntax.

In our formalism for describing relations and events (see section 3.1) there is provision for including a time reference in a relation. The sentence "Harriet saw the film last week." might be represented as

(#SEE :HARRIET :FILM :TIME23)

where :TIME23 is an arbitrary name for a structure describing the time reference "last week". The semantic programs for dealing with time can be described in three parts -- the form of structures used to represent time, the way those structures are created, and the way they are used in understanding and deduction.

A. Time Semantic Structures

For the purposes of our BLOCKS world, we have treated only a simple part of the overall range of time references in English. In particular we have dealt only with references to actual events which have happened in the past or are occurring in the present, without dealing with the many varieties of

future events, possible events, conditional events, etc. With this simplification the system can use a simple linear time scale (like a clock), relating all events to specific numerical times. This does not mean that a single event must occur at a single time -- it may continue for a period of time during which other events are occurring.

English makes a clear distinction between events which are thought of as occurring at a particular time, and those which are pictured as continuing over an interval. This contrast is expressed both in the choice of verbs and in the shape of the VG containing the verb.

Verbs like "like", and "know", are inherently progressive. They express a relationship which continues over a period of time. Verbs like "hit", and "write" are not progressive, but indicate the completion of an action as a whole. Of course, this action also involves a process, and there is a way to express this aspect by using a tense PRESENT IN... The sentence "I broke it." is not progressive, giving the feeling of a single momentary act. "I was breaking it." emphasizes the process of breaking, to which other events can be related.

In the present tense, the distinction is clear. The present of a progressive verb has the expected meaning, as in "I know your name." With a non-progressive verb, there is a special meaning of habitual or repeated action, as in "I break bottles." In order to produce the meaning usually considered "present",

the verb group must be PRESENT IN PRESENT, as in "I am breaking bottles."

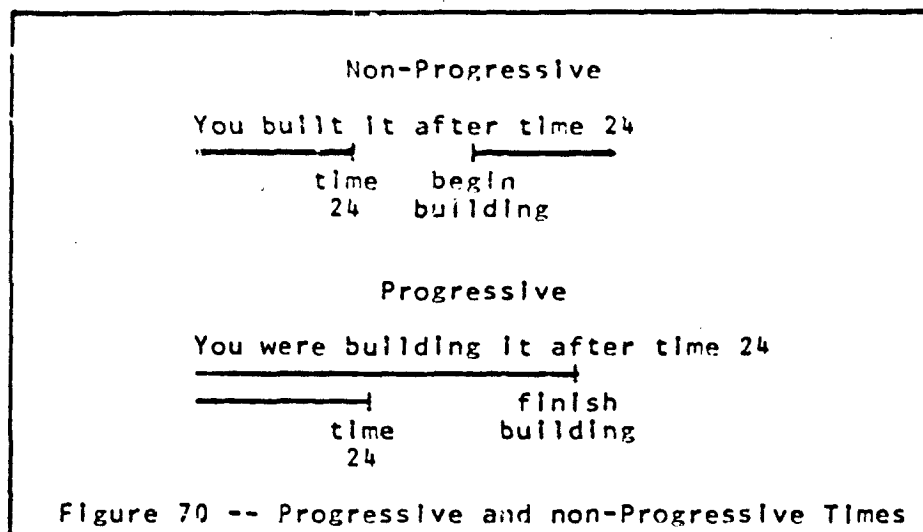
Ambiguities can arise from verbs which are both progressive and non-progressive. The question "Did the red block touch the green one while you were building the stack?" has two interpretations. One means "Was it in contact during that time?", while the other asks "Did it make contact during that time?" If the verb were replaced by "support", only the analog of the first meaning would be valid, while "hit" would involve the second. The representation for time references must take this progressivity into account in trying to interpret time modifiers.

The representation used for time has four elements: the tense, an indicator for progressive, a starting time limit, and an ending time limit. Either or both of the limits may be omitted. Some examples of sentences and their corresponding structures are shown in Figure 69.

A supports B	(PRES)	T	:NOW	:NOW
A supported B before time 23	(PAST)	T	NIL	23
A hit B before time 23	(PAST)	NIL	NIL	23
You built it after time 24	(PAST)	NIL	24	NIL
You were building it after time 24	(PAST)	T	24	NIL

Figure 69 -- Time Semantic Structures

The difference between the last two examples in Figure 69 can be visualized by drawing a time line:



A non-progressive action must begin after the start time, and end before the end time. A progressive one begins before the start time and ends after the end time. The TSS for "you hit it during event 23" (assuming event 23 began at time 3 and ended at 7) would be

(PAST) NIL 3 7

i.e. the hit began after event 23 started and ended before it ended. The sentence "you were hitting it during event 23" would be:

(PAST) T 7 3

i.e. the hitting began before event 23 was over, but ended after it had begun. This covers all ways of having the two events overlap. The definitions of the relating words like "during" and "before" do not have explicit mention of this distinction, but the semantic analysis programs take into account whether the verb and VG are progressive in setting up the TSS.

B. Setting up Time Structures

Time Semantic Structures are associated with clauses, and a new one is generated each time a clause is parsed. Its elements are determined by different aspects of the clause structure -- the tense depends on the form of the VG, the progressivity depends on both the tense and the specific verb, and the limits are set by modifiers such as bound clauses, adverbs, and time NGs as well as by the tense.

No analysis is done until after the VG is parsed and the tense established. Some types of secondary clauses such as ING, SUBING, TO, and SUBTO do not indicate a tense. There is a potential ambiguity in determining the time reference. "The man sitting on the table baked the bread." might indicate that the man was sitting on the table when he baked it, or that he is sitting on the table now.

Unless there is a specific reference (like "the man sitting on the table yesterday...") the system should take both possibilities into account and resolve them as it would an ambiguity caused by multiple senses of words. The current system does not do this, but uses a simplifying heuristic. If the secondary clause involves PAST, and is embedded in a PAST MAJOR CLAUSE, the two times are assumed the same unless specifically mentioned. If the secondary clause has no tense, it is assumed PRESENT. If it is PAST, but imbedded in a PRESENT MAJOR CLAUSE, the system checks the time reference of the

previous sentence. If this is PAST, the new one is assumed to be the same (including whatever modifiers, limits, etc. applied). If not it sets up a general time structure for PAST, with no beginning limit, and an end limit of :NOW. A PRESENT tense TSS is represented by the single atom :NOW, which is treated specially by the programs, and is often deleted from relations which interrogate the current state of the data base (see below). It can be applied only to progressive verbs and tenses (no provision exists for understanding habitual action).

Modals are treated like present tense as far as establishing time references. A more complete system would account for future, different types of modals, more complex tenses, and would involve heuristics for finding the referents of multiple tenses like "He will have been going to go immediately for a month by Tuesday."

The start and end limits are set by modifiers. Adverbs like "yesterday" and TIME NG's like "the week he arrived" set both limits. This can also be done by bound clauses like "while you were building the stack" or PREPGs like "during the flood". Other clauses, prepositions, and groups set only the start limit (like "after you hit it", "after the war") while others (like "before" and "until") set the end limit. In the current system the event being referred to in the modifier is assumed to be known along with its exact time (it must be in the past.) The exact beginning and ending time are used in setting the limits.

The question "Did you pick it up while you were building the stack?" is answered by first finding the event of building the stack (using a TSS for PAST tense with no other limits), then using the beginning and ending of that event as limits for the TSS in the relation #PICKUP.

There are discourse phenomena which involve time reference. First, there are specific back-references with words like "then" and phrases like "at that time". The system keeps track of the major time reference of the previous sentence, and substitutes it in the current sentence whenever such phrases are used. This time is also carried forward implicitly. Consider "Did you pick up a red block while you were building the tower?" "No." "Did you pick up a green one?" In this sequence, the second question involves a specific time interval although it is not mentioned again. Whenever there are two successive PAST sentences and the second does not have any explicit time reference, the previous TSS is used. Long dialogs can appear in which the same time interval is used throughout, but is mentioned only in the first sentence.

C. Use of TSS

So far, all of our discussion has involved the clause with its verb group and time modifiers. But in making use of time information we must handle other units as well. The sentence "The man sitting on the table baked the bread." has two meanings, but the point would have been identical for "The man

on the table baked the bread." The qualifying prepositional phrase "on the table" does not refer to time, but can be interpreted either as meaning "on the table now" or "on the table then". Adjectives can be affected similarly. Consider the sentences:

- a. Many rich men made their fortunes during the depression.
- b. Many rich men lost their fortunes during the depression.
- c. Many rich men worked in restaurants during the depression.

The first clearly means "men who are now rich", the second "men who were rich", and the third might have either interpretation. The adjective "rich" involves an implicit time reference, as does any adjective which describes a state which can be true of an object at one time, but false at another. Nouns can also involve states which are changeable, and the problem would be identical if "rich men" were replaced by "millionaires".

In a traditional transformational approach, this would be used to show that even a simple phrase such as "a rich man" or "millionaires" is generated by a series of transformations. The possibility of two meanings is accounted for by two different deep structures, involving sentences corresponding to "The men were rich." and "The men are rich." This leads to a syntactic theory in which the simplest sentence may involve dozens of such transformations, to account for each noun, adjective, preposition, etc. The parser must be able to handle all of

these details using syntactic information.

in our approach, these can be seen as semantic ambiguities which arise within a single syntactic structure. Part of the semantic definition of the word "millionaire" (or "student", "bachelor", etc.) involves a reference to time. Within the language for writing semantic definitions, there is a special symbol *TIME. Whenever the program for the meaning of a word in the dictionary is called, the semantic system will have determined the appropriate Time Semantic Structure (or structures) and have assigned a value to this symbol accordingly. If the time reference is ambiguous, the definition will be called once for each possibility. The noun "millionaire" might be defined:

```
(NMEANS ((#PERSON) ((#IS *** #PERSON)
                    (#POSSESS *** $1,000,000 *TIME))))
```

Notice that not every relation involves time. Being a #PERSON is assumed to be a permanent characteristic. If the time is PRESENT (indicated by the TSS :NOW), the system deletes the time reference, so PLANNER will receive the expression (THGOAL (#POSSESS \$?X1 \$1,000,000)), where \$?X1 is the variable assigned to the object being described. If the sentence were "During the war, many millionaires worked in restaurants.", the time reference of the sentence would be a structure like ((PAST) NIL 1941 1945), and the PLANNER expression for "millionaire" would include:

```
(#POSSESS $?X1 $1,000,000 ((PAST) NIL 1941 1945))
```

A different theorem would be used for this case, since it cannot look directly into the data base to see what the person has, but must look into its past "records" to reconstruct the information. In our programs, a record is kept of when and where objects have been moved, so theorems can determine the location of any object at any time in the past.

Since adjectives can be defined with NMEANS, they are treated identically. PREpositions and verbs are usually defined with CMEANS, which has the same conventions. The symbol *TIME can appear in the PLANNER description in the definition, and is deleted if the applicable time is :NOW, and replaced with the TSS otherwise. The time applicable to anything but a clause is that of the clause closest above it in the parsing tree. This is only an approximation, and does not take into account ambiguities such as illustrated in sentence c. above. In fact, a PREP or NG can have its own time reference, as in "a former millionaire", "many future students", "my roommate last year", "the man on the table yesterday". This is one of many places where the current semantic system needs to be extended by making the analysis more general. It seems that this could be done within the framework of the current system.

4.2.9 Semantics of Conjunction

The semantic system does not handle conjunction as generally as does the parser. A few cases have been dealt with in a simplified way -- noun groups, adjectives, RSQ clauses, and MAJOR clauses which are not questions. The distinction between "and" and "but" is ignored.

With MAJOR clauses, the conjunction must be "and", and the components are processed as if they were completely separate sentences, except that the response ("OK." for IMPERatives, and "I UNDERSTAND." for DECLARatives) is suppressed for all but the last. The system will not accept sentences joined with "or", or "nor", and will misunderstand compounds which cannot be separated into individual actions (e.g. "Build a stack and use three cubes in it.")

Noun groups can be connected with "and" wherever they appear, and with "or" if they are part of an argument to a command (like "Pick up a cube or a pyramid."). An OSS is built with the semantic markers of the first constituent NG, the conjunction itself, and a list of the OSS for the components. If all of the components are DEFINite and the conjunction is "and", the conjoined NG is definite, and its REFERent is the union of the referents.

The use of the conjoined OSS depends on its place in the sentence. If it is the object or subject of a verb or preposition, the definition of that verb or preposition can

check explicitly for conjoined structures and treat them specially. For example, "touch" can be defined so that the sentence "A and B are touching." will be represented as (THGOAL (#TOUCH :A :B)). If there is no special check, the system assumes that the desired object is the list of referents. "A and B support C." would produce (THGOAL (#SUPPORT (:A :B) :C)). If the first element of the PLANNER expression (usually the name of a predicate) has a property MULTIPLE on its property list, the system modifies this to create the expression:

```
(THAND(THGOAL(#SUPPORT :A :C))
      (THGOAL(#SUPPORT :B :C)))
```

If the conjoined NG is one of the arguments to a command, the theorem TC-CHOOSE will choose the specific referents. If the conjunction is "and", it will combine the referents for each of the components in a single list. If it is "or", it will first choose according to the first constituent, then if a failure backs up to the choice, it will try the second, third, etc. It does not look at the various choices in advance to decide which is most appropriate for the task being done.

The other units which can be combined with "and" and "or" are the adjective and RSQ clause. The semantic structure for the conjoined unit is a list whose first element is the conjunction, and the rest are the individual interpretations for the constituents. In using these to modify an OSS, the system combines all of the descriptions with THOR or implicit THAND. For example, "a block which is in the box and is red" becomes:

```
(THGOAL(#IS $?X #BLOCK))
(THGOAL(#IN $?X :BOX))
(THGOAL(#COLOR $?X #RED))
```

while "a red or green block" becomes:

```
(THGOAL(#IS $?X #BLOCK))
(THOR(THGOAL(#COLOR $?X #RED))
      (THGOAL(#COLOR $?X #GREEN)))
```

This could easily be extended to other modifiers such as preposition groups. Many other types of conjunction could be handled without major changes to the system, usually by adding two bits of program. One would create a conjoined semantic structure appropriate to the unit, and the other would recognize it and take the appropriate action for its use.

Whenever the constituents of a conjoined structure are ambiguous, the resultant structure simply multiplies the ambiguity, taking all possible combinations of interpretations.

4.2.10 More on Ambiguity

Section 4.1 described how the number of interpretations of an ambiguous sentence can be reduced through the use of semantic markers, and selection restrictions associated with verbs, adjectives, and prepositions. This section will describe the mechanism for producing multiple interpretations, assigning plausibilities to them, and resolving the ambiguities through discourse heuristics and interaction with the user.

Any word of the classes ADV, ADJ, NOUN, PREP, PRON, PROPN, VB, CLASF, or PRT can introduce an ambiguity into the semantic interpretation. The remaining classes (such as NUMBER and DETERMINER) have very limited definitions, and are handled differently.

In general, a word is expected to produce a list of semantic structures, based on its definition, and the other lists of semantic structures to which it is related. NOUN, PRON, and PROPN set up lists of Object Semantic Structures. ADJ, ADV, and CLASF take one of these lists, and produce a new list adding the modification (and possibly eliminating anomalous combinations). The VB, PREP, and PRT (in conjunction with VB) set up lists of Relation Semantic Structures, and other classes can modify these lists.

Any of these definitions can involve special programs for producing the list of structures. For example the SMIT program is used for analyzing pronouns like "they" and "it". It

contains a complex set of heuristics and syntactic criteria to find the possible referents of a pronoun and set up an interpretation for each one. For simpler words, the functions NMEANS and CMEANS have ways to deal with multiple senses of a word.

First, they both take as an argument not a single definition, but a list of definitions, each with its own semantic markers, PLANNER expressions, etc. Second, each of them has mechanisms for looping through each of the relevant "input" lists, to produce multiple interpretations. If the subject and object of a clause each have two interpretations, and the verb has three senses, all twelve combinations will be tried, and the resulting list of interpretations for the clause will contain as many of them as pass through the semantic marker "filters". This is all done by the function CMEANS without mention in the definitions. Third, the definition functions have optional ways to establish a "plausibility" rating and paraphrase for each meaning.

In the dictionary, the word "on" has a semantic definition which uses the function #ON, which contains:

```
(CMEANS (((#PHYSOB)) ((#PHYSOB))) (#ON #1 #2 *TIME)
          NIL
          (200 (ANYWHERE ON TOP OF)))
  (((#PHYSOB)) ((#PHYSOB))) (#SUPPORT #2 #1 *TIME)
    NIL
    (0 (DIRECTLY ON THE SURFACE)))
  (((#PLACE)) ((#PHYSOB))) (#ON #1 #2) NIL))
```

This contains three different senses. The third definition will never conflict with the first two, since it insists that the subject be a place rather than a physical object. The first two, however can be applied to the same objects, and therefore can be involved in an ambiguity. Each of them is given an additional argument, to help resolve ambiguities. This argument appears after the standard semantic filters, PLANNER relation, and semantic markers for the relation. It is composed of two elements -- a LISP form to be evaluated for a "plausibility", and a paraphrase of the meaning. In this example, the plausibilities are simply numbers. They could just as easily be a form like (PLAUSCHECK), which would call a special program (defined by the user) which could do arbitrary calculations in order to decide on a plausibility. This might involve PLANNER deductions, checks on the exact syntax of the sentence, or other schemes such as looking through a network or other model in order to decide which meaning fits best with the the other words in the sentence and the subject being discussed.

As a semantic structure is built, it takes on the sum of the plausibilities of its components, as its own plausibility. No "pruning" is done to eliminate obvious low plausibilities, although this would be a simple addition to the program. All interpretations are carried along until the sentence is totally parsed.

If the sentence is a command, the system tries to carry out

the most plausible interpretation. If that fails, it tries the next, and so on until one succeeds or a total failure causes the system to respond "I CAN'T". Questions are handled more completely. The system orders the interpretations by plausibility and finds the answer for the most plausible. It then tries again to answer it, using only information mentioned in the previous sentence and its answer. If it succeeds in finding the same answer, it deducts 500 from the plausibility, since it is unusual to ask a question to which the answer was just given, either explicitly or implicitly. If the information in the previous sentence is not sufficient to answer it, the system then tries to answer using only information which has been mentioned previously in the discourse. If this succeeds it deducts 200. If the plausibility is higher than that of the next interpretation by a large enough margin (a factor set by the user and called TIMID) it gives the answer as found. If not, it saves the answer and repeats the process for the next interpretation. After all interpretations have been processed, the answers are checked to see if they are identical. In this case it doesn't matter which interpretation is intended, and the system simply gives the answer. Finally, if there are differing answers, the user must be asked what he meant. Associated with each interpretation is a list of those places where it differed from others. This is produced automatically by each program which accepts multiple definitions (such as NMEANS and CMEANS).

Each difference is marked by two atoms -- one whose properties indicate the place in the sentence where the ambiguity was produced, and the other indicating the meaning selected. In sorting out the ambiguities, the system looks for two such structures with the same first atom but different second ones. The first can be used to decide what phrase is questionable, while the second atoms carry the paraphrases. Special care is taken to make sure that the same sentence interpretation does not involve two different interpretations of a single element (like "it").

Faced with an unresolvable ambiguity, the system looks through the list of interpretations for a conflict, then generates a response like:

I'M NOT SURE WHAT YOU MEAN BY "ON TOP OF" IN THE PHRASE "ON TOP OF GREEN CUBES".

DO YOU MEAN:

- 1 - DIRECTLY ON THE SURFACE
- 2 - ANYWHERE ON TOP OF ?

The response (a typed number) indicates which is meant, and all interpretations which involve the other meanings (there can be more than 2 associated with a single ambiguity) are eliminated. If there are still conflicting interpretations, another ambiguity is selected and the process is repeated until all those which have not been eliminated give the same answer, and it can be used as a response.

For interpreting statements, much more subtlety is needed. In general the choice between interpretations depends on how

"reasonable" an interpretation is. If we see "The book fell off the table because it was slanted.", we avoid believing that "it" refers to the book, or that it was "slanted" by the author's bias. It would not be reasonable for either of these to be the reason for falling off of a table.

Part of the interpreter which looks at declarative statements must be able to evaluate how "surprising" an interpretation is, and to choose the one which fits best into the normal expectations, including the hearer's knowledge about the world. This is not easy to formalize, and was not attempted in this project. It could be extended to lower levels to check sub-units for reasonableness as they are formed, to avoid carrying multiple interpretations through the analysis of the entire sentence. It seems clear that people do this, and a truly intelligent language-understanding program must do so as well.

4.2.11 To Be and To Have

The verbs "be" and "have" are two of the most common words in English, and have a complex variety of uses. They appear in the system in two ways. First, in the grammar they are treated specially since they can occur as auxiliary verbs, as in "I would have been going". In this use, they do not add any semantic information except for helping to determine features of the VG, such as its tense. Their other use is as main verbs in clauses like "Do you have a match?" and "He is wrong." As a main verb, "be" is handled specially in the grammar since it can enter into constructions such as "there is" which do not appear with any other verb. However, the semantic analyzer does not know anything special about "be" and "have". Their meaning is included in their definitions, which are called as programs just like any other verb definitions.

A. Be

The use of "be" depends on the specific constellation of objects and complements in the clause. The definition is a program containing about 40 lines of LISP, which handles those meanings relevant to the BLOCKS world (for example, it cannot deal with a "role-playing" meaning, like "Laurence Olivier was Hamlet.")

Sentences with the feature THERE, involving a construction like "there is" are represented by the PLANNER expression (#EXISTS #1 *TIME). This attaches the correct time, and might

be important for objects which can be created and destroyed, as in "Was there a stack...?".

The other meanings of "be" involve INTensive clauses which contain an object and a complement. One definition checks for COMPQ questions like "What color is the block?", to generate a PLANNER expression (#COLOR :BLOCK \$?X1). If the complement is a definite NG, as in "Is the green block the biggest object?" or "What is the biggest object?", the referent will have already been determined, and is inserted in a PLANNER expression (THAMONG *** (QUOTE(:OBJ))), where :OBJ is the referent. This can function in two ways. If the subject is also definite, as in the first example, the *** will be replaced by its referent, and the statement will succeed only if the two are identical. If the subject is indefinite, the THAMONG statement will cause it to be assigned to the same referent as the complement.

If the complement is a PREPG or a complex ADJG, like "bigger than a breadbox", "be" is only serving as a place-holder which can accept a time reference. The semantic interpreter in dealing with a phrase like "on the table" in "Is the block on the table?" has already set up a relation of the form (#ON :BLOCK :TABLE) which includes the appropriate time reference. In this case, the "be" program simply takes the RSS produced for the complement, and uses it as the semantic interpretation of the clause.

The other possibilities for the complement are an

indefinite NG, a simple ADJG (e.g. a single adjective), or a new word. In the case of a NG, the complement NG contains additional information to be ascribed to the subject, as in "a large object which is a red block". The PLANNER description of the complement is stripped from its OSS, and appended to the PLANNER description of the subject. If the subject is definite, as in "Is the biggest thing a red block?", the referent is known, and can be plugged into the PLANNER description of the complement to see if the description applies. This is done using a pseudo-concept called #HASPROP which triggers the mechanisms in the semantic interpreter.

If the complement is a simple ADJG, the ADJG semantic specialist creates its OSS by taking the OSS for the subject, stripping away the PLANNER description, and using the rest as a skeleton on which to place the PLANNER expression produced by the adjective. Once this is done, it can be treated exactly like an indefinite NG.

Finally, if the subject or complement is a new word (as in "A frob is a big red cube." or "A big red cube is a frob.") a new definition is created using the function #DEFINE. The definition must be in the form of an indefinite NG, and the new word is assumed to be a noun. The semantic definition created for the noun contains the OSS which was created for the defining NG, and sets this OSS up as the meaning of the noun when it is used.

B. Have

The definition of "have" is also used to handle the possessive. For the limited subject matter (and for much of English) this is a good approximation. There are cases where it does not apply -- "the painting which John has" is not necessarily the same as "John's painting." The preposition "of" also makes use of the same definition. A more complete treatment would distinguish between the three, and this would involve only simple changes to the semantic programs.

The interesting thing about "have" is that it is not used to indicate a few different relationships, but is a place-marker used to create relationships dependent on the semantic types of the objects involved. "Sam has a mother." can be represented (#MOTHER-OF X SAM), "Sam has a friend." is (#FRIEND X SAM), "Sam has a car." is (#OWN SAM CAR), "Sam has support." is (#SUPPORT X SAM), "Sam has a hand." is (#PART SAM HAND), etc. The definition of "have" (or the possessive, or "of") does not include within itself all of these different relations. A few interpretations (like have-as-part, owning, or having in physical possession) can be reasonably considered distinct meanings of "have", and are included in its definition. The others, such as "mother" and "support" really are determined by the subject and object. Some systems use this fact to find the meaning of special phrases like "client's lawyer" without doing syntactic analysis (see section 2.5). Our system uses a

different method, allowing a word to be defined as a #ROLE.

"Mother" might be defined as:

```
(NMEANS((#PERSON #ROLE)
          ((#PERSON ***)
           (#MOTHER-OF *** ?)
           (#ROLE((#PERSON))(#MOTHER-OF #1 #2))))
```

There are two new things in this definition. First, the semantic marker #ROLE is added to indicate the type of definition. Second, a role definition is included. It contains a semantic filter for objects which can be used in the relation (in this case those which could have a mother), and a PLANNER statement indicating the relation (in the same syntax used by CMEANS). If the word "mother" is used in a phrase like "Carol's mother" or "Carol has a mother" or "the mother of Carol", the system will insert the right OSS to produce the PLANNER description (#MOTHER-OF \$?X1 CAROL) if "mother" appears in any other form, the OSS will contain (#MOTHER-OF \$?X1 ?) which will be satisfied in a PLANNER goal if X1 is the mother of anyone at all.

Through the #ROLE mechanism, arbitrary relationships can be expressed with "have" (or "of", or possessives) without bloating its definition. There could be more than one #ROLE assigned to a word as well. For example "painting" would involve different roles for "Rembrandt's painting" "George Washington's painting by Stuart", "the Modern Museum's painting.", etc.

4.2.12 Additional Semantic Information

A. Using Clauses as Objects

In order to interpret a sentence like "Find a block which is taller than the one I told you to pick up." the system must use a clause ("you to pick up") as the object of a verb ("tell"). It generates a pseudo-object of the type #EVENT, and creates an OSS for that object. In the example mentioned, the clause "you to pick up" would have produced the RSS:

```
((NG1 (#PICKUP NG1 ((PAST) NIL NIL NIL))      NIL) (0) )
  rel                PLANNER expression      neg markers
```

Figure 71 -- RSS for "you to pick up"

NG1 is an OSS describing the object "the one", which the system has set up as the object of the clause, and has interpreted as "block". The program SMCL4 takes this structure and produces a corresponding OSS:

```
( ( ((EVX1) 0 (THGOAL (#PICKUP $?EVX1 $?X1 ((PAST) NIL NIL NIL))
  (THUSE TCTE-PICKUP)))
  (0 #EVENT #THING)
  (#THING))
EVX1
(1 INDEF NIL)
NIL)
```

Figure 72 -- OSS for "you to pick up"

A variable was generated for the event, of the form EVXn, and a new PLANNER expression for the event was generated, including the event name as the second element. The reason for putting it second is technical, and should be changed someday for programmer convenience and consistency with the scheme

described in Section 2.1. In the expression, the name of the OSS is replaced with its associated variable (in this case \$?X1) since the new structure will be used as part of the description of that object. The recommendation list includes the theorem which is designed to deal with expressions involving time and event-names, and is put in by the system. In working with the rest of the sentence, this resultant OSS can be used just like any other OSS, as an object of a verb, preposition, etc.

When PLANNER evaluates the expression, it may have the event already stored away, or it may have to deduce that it happened by looking at other events. This is handled by the theorem TOTE-PICKUP, and the name of the resultant event is the value which is assigned to the variable EVX1.

B. Types of Modification

There are a variety of ways in which a modifier can affect the meaning of the phrase or clause it modifies. Since the definition is a program, the user has great freedom to use different types of modification. A time modifier like "now" or "then" will modify the Time Semantic Structure associated with the clause, an adverb like "quickly" may set up a new relation such as (#SPEED \$?EV1 #FAST) using the name of the event, while others may make changes directly to the relation being constructed. The semantic structures previously built can be analyzed and modified by an arbitrary function which suits the meaning of the modifier. One special facility exists for making

substitutions within an expression. If the PLANNER expression of a CMEANS or NMEANS definition is of the form (#SUBST a1 a2 b1 b2...), the effect will be to modify the existing semantic structure by substituting the atom a2 for a1, b2 for b1, etc. No new expression is added to the PLANNER description. The word "move" might be defined using:

```
(CMEANS((((#ANIMATE))((#MANIP))) (#PUT #2 LOC *TIME) (#MOVE)))
```

This indicates that moving is done by an animate object to a manipulable object, and involves putting it at the place "LOC". The atom LOC would be given a OSS indicating an unknown place. The resulting RSS has the semantic marker #MOVE. The sentence "Move a block." would create a goal (#PUT NG1 LOC), where NG1 is a description of "a block". The theorem for #PUT could then choose a block and place. If the sentence is "Move a block into the box.", the final result should be (#PUTIN NG1 :BOX). The modifying phrase makes a major change in the internal representation of the meaning.

This change can be done by defining "into" to include among its meanings:

```
(CMEANS((((#MOVE))((#BOX))) (#SUBST #PUTIN #MOVE #2 LOC) NIL))
```

If a PREPG with the preposition "into" modifies a clause with the semantic marker #MOVE, and the object of the preposition has the marker #BOX, then the definition applies. The RSS for the clause is changed by substituting #PUTIN for #MOVE, and the object of the preposition for #LOC. The special symbols #1, #2,

#3, ***, and *TIME are treated as they would be in a normal CMEANS or NMEANS definition, being replaced by the appropriate object.

C. Using Evaluation In CMEANS and NMEANS

Although every definition has the power to use programs, definitions using the standard forms CMEANS and NMEANS are forced into a rather rigid syntax which does not have a procedural character. To avoid this, there is an extra level of evaluation. If the PLANNER portion of a definition is of the form (#EVAL s) where s is any LISP atom or s-expression, the form will be EVALled before the description is used in the definition, and its value used instead. This value will undergo the usual substitutions for #1, #2, *TIME, etc. This feature is of particular use in capturing the semantic regularities of the language by using auxiliary functions in defining words. For example, color adjectives like "red" and "blue" share most of their characteristics. They apply to physical objects, involve a relation with #COLOR, etc. Rather than define them separately, we would like a single function #COLOR which needs only to have the exact color specified. The dictionary definition of blue would then be (#COLOR #BLUE). The function #COLOR can be defined in LISP:

```
(DEFUN #COLOR FEXPR (A)
  (NMEANS((#PHYSOB) (#EVAL (LIST(LIST (QUOTE #COLOR)
                                       (QUOTE ***)
                                       (CAR A)))))))
```

When (#COLOR #BLUE) is evaluated, the #EVAL will produce the form ((#COLOR *** #BLUE)), which will then be used by NMEANS in the usual way.

As another example, the word "grasp" can be used to mean #GRASPING (an object being held) or #GRASP (the action of closing the fingers around it). The difference depends on whether the VG is progressive or not. The function (PROGRESSIVE) finds out whether the clause is progressive, by looking at the verb and the tense. The definition of "grasp" can be:

```
(CMEANS((((#ANIMATE))((#MANIP)))
  (#EVAL (COND ((PROGRESSIVE) (QUOTE(#GRASPING #2 *TIME)))
    (T (QUOTE (#GRASP #2 *TIME))))) NIL))
```

D. Some Interesting Problems

There are many areas in which the semantic analysis needs to be refined and expanded. The system does not pretend to contain a complete analysis of English, but is rather an illustration of how many aspects of semantics could be handled. This section describes a few places where modification might begin.

1. Definite Determiners

in our system, a definite noun phrase is interpreted as referring to a unique object or set of objects known to the hearer. In more general language use, definiteness is often used to convey new information. The phrase "my brother who lives in Chicago" can be said to someone who is not aware I have

a brother, and the effect is to inform him that indeed I do, and to tell him where this brother lives. Other nouns can describe "functions", so that "the title of his new book", or "my address", are allowable even if the hearer has not heard the title or address, since he knows that every book has a unique title, and every person an address. Superlative phrases like "the tallest elephant in Indiana" also refer to a unique object, even though the hearer may not have seen or heard of this object before.

Cases such as these can lead to problems of referential opacity. If your name is "Seymour", and I say "Excuse me, I've never heard your name.", it does not imply that I have never heard the name Seymour. The sentence "I want to own the fastest car in the world." does not have the same meaning if we replace the NG with its current referent -- I don't want whichever car it is that happens to be fastest right now.

These and other such problems need to be handled in the programs for interpreting a definite NG, using syntactic, semantic, and world knowledge.

2. Verb Tenses

The current system implements only a few of the possible tenses -- PRESENT, PAST, PRESENT IN PRESENT, PRESENT IN PAST, PAST IN PAST, and an elementary form of the MODAL "can". Section 4.2.8 described some of the problems which can be involved in time reference, and a deeper analysis is needed to

account for interactions between the order of phrases and the possibilities for time reference. The modals, conditionals, subjunctives, etc. need to be handled. This may demand a version of PLANNER which can temporarily move into a hypothetical world, or which has more power to analyze its own theorems to answer questions involving modals like "can" and "must".

3. Conjunction

Only the most elementary problems in conjunction have been dealt with in the current system. Many conjoined structures do not yet have semantic analyzer programs, and no subtlety is used in deciding on the meaning of words like "and". "And" can be used to indicate temporal sequence ("We went to the circus and came home.") causality ("We saw him and understood."), as a type of conditional ("Do that again and I'll clobber you!"), in specification of how to do something ("Be a friend and help me."), etc. Understanding these uses will be related to the discourse problem of the ordering of sentences. For example, "The light is on. He's there." indicates a chain of reasoning.

In addition, no attempt has been made to disambiguate nested structures like "A and B or C", or "the old men and women." Syntactic criteria are not sufficient for these distinctions, and a powerful semantic program will have to be used to ask "which interpretation makes more sense in this case".

4. Non-syntactic Relations

There are some places in English where the relation between a set of words is not indicated by syntactic clues, but is largely based on semantics. One example is chain of classifiers before a noun. In "strict gun law", the law is strict, but in "stolen gun law", the gun is stolen. It is possible to combine long strings like "a helical aluminum soup pot cover adjustment screw clearance sale", in which a large amount of semantic information must be combined with the ordering to find the correct interpretation. The current system handles classifiers by assuming that they all separately modify the head. This needs to be changed, to use both the semantic markers, and complex deductions to find the real relationships.

4.3 The Semantics of Discourse

In section 4.1, we discussed the different types of context which can affect the way a sentence is interpreted. In this section we will describe the specific mechanisms used by our program to include context in its interpretation of language. We have concentrated on the "local discourse context", and the ways in which parts of the meaning of a sentence can be referred to by elements of the next sentence. For example, pronouns like "it" and "they" can refer to objects which have been previously mentioned or to an entire event, as in "Why did you do it?". The words "then" and "there" refer back to a previous time and place, and words like "that" can be used to mean "the one most recently mentioned", as in "Explain that sentence."

In addition to referring back to a particular object, we can refer back to a description in order to avoid repeating it. We can say: "Is there a small grey elephant from Zanzibar next to a big one?". Sometimes instead of using "one" to avoid repetition, we simply omit part of a phrase or sentence. We can reply to "Would you like a corned-beef sandwich?" with "Bring me two." or we can respond to almost anything with "Why?" In these examples, the second sentence includes by implication a part of the first.

These are not really discourse features, since they can appear just as well in a single sentence. In fact, there are some sentences which would be almost impossible to express

without using one of these mechanisms, such as: "Find a block which is bigger than anything which supports it." These mechanisms can be used to refer back to anything mentioned previously, whether in an earlier sentence of the speaker, one of the replies to him, or something occurring earlier in the same utterance.

4.3.1 Pronouns

First we will look at the use of pronouns to refer back to objects. Since our robot does not know any people other than the one conversing with it, it has no trouble with the pronouns "you" and "I" which always refer to the two objects :SHRDLU and :FRIEND. A more general program would keep track of who was talking to the computer in order to find the referent of "I".

When the NG program in the grammar finds a NG consisting of a pronoun, it calls the program which is the definition of that pronoun. The definitions of "it" and "they" use a special heuristic program called SMIT, which looks into the discourse for all of the different things they might refer to, and assigns a plausibility value to each interpretation. If more than one is possible, they are carried along simultaneously through the rest of the sentence, and the ambiguity mechanism decides at the end which is better, including the last-resort effort of printing out a message asking for clarification. If SMIT finds two different interpretations, and one is chosen because of a higher plausibility, the system types out a message to inform us of the assumption made in choosing one interpretation, as in Sentence 3 of Section 1.3:

BY "IT", I ASSUME YOU MEAN THE BLOCK
WHICH IS TALLER THAN THE ONE I AM
HOLDING.

If a response from the user is needed, the request is typed in the same format as the message used for other ambiguities, as

in sentence 24 of Section 1.3. In the case of sentence 3, it would be:

I'M NOT SURE WHAT YOU MEAN BY "IT" IN
THE PHRASE "PUT IT INTO THE BOX"

DO YOU MEAN:

1 - THE BLOCK WHICH IS TALLER THAN THE
ONE I AM HOLDING

2 - THE ONE I AM HOLDING ?

A simple transformation is used to switch "you" with "I", and make the corresponding verb changes, and the words are borrowed directly from the input sentences.

In our discussion of pronouns, we will use "it" as typical. In most cases, "they" (or "them") is treated identically except checking for agreement with plural rather than singular. The pronouns "he" and "she" never occur in our limited subject matter, but they would be treated exactly like "it", except that they would make an extra check to see that their referent is in fact animate and of the right gender.

The first thing checked by SMIT is whether "it" has already appeared in the same sentence. We very rarely use the same pronoun to refer to two different objects in the same sentence, so it is generally safe to adopt the same interpretation we did the first time. If there were several possible interpretations, the system is careful not to match up one interpretation from one occurrence of "it" with a different one from another occurrence in building an overall interpretation of the sentence.

Similarly, if "it" was used in the previous sentence, it is likely that if used again it will refer to the same thing. In either of these cases, SMIT simply adopts the previous interpretation.

Next, a pronoun may be inside a complex syntactic construction such as "a block which is bigger than anything which supports it." English uses the reflexive pronouns, like "itself" to refer back to an object in the same sentence. However, if in going from the pronoun to the referent on the parsing tree, it is necessary to pass through another NG node, an ordinary pronoun like "it" is used, since "itself" would refer to the intermediate NG. Notice that if we replaced "it" by "itself" in our sentence, it would no longer refer to the block, but to "anything".

SMIT looks for this case and other related ones. When such a situation exists, the program must work differently. Ordinarily, when we refer to "it" we have already finished finding the referent of the NG being referred back to, and "it" can adopt this referent. In this case, we have a circle, where "it" is part of the definition of the object it is referring to. The part of the program which does variable binding in relating objects and clauses is able to recognize this, and treat it correctly by using the same variable for "a block" and "it".

The pronoun may also refer to an object in an embedded clause appearing earlier in the same clause, as in "Before you

pick up the red cube, clear it off." SMIT looks through the sentence for objects in such acceptable places to which "it" might refer. If it doesn't find them there, it begins to look at the previous sentence. The pronoun may refer to any object in the sentence, and the meaning will often determine which it is (as in our example about the demonstrators in the Preface). We therefore cannot eliminate any of the possibilities on syntactic grounds, but can only give them different ratings of "plausibility". For example, in Section 4.2.5 we discussed the importance of a "focus" element in a clause. "It" is more likely to refer to the previous focus than to other elements of the clause. Similarly, the subject is a more likely candidate than an object, and both are more likely than a NG appearing embedded in a PREPG or a secondary clause.

The system keeps a list of all of the objects referred to in the previous sentence, as well as the entire parsing tree. By using PROGRAMMAR'S functions for exploring a parsing tree, SMIT is able to find the syntactic position of all the possible references and to assign each a plausibility, using a fairly arbitrary but hopefully useful set of values (for example we add 200 for the focus element beyond what it would normally have for its position as subject or object). In order to keep the list of the objects in the last sentence, our semantic system has to do a certain amount of extra work. If we ask the question: "Is any block supported by three pyramids?", the PLANNER expression

produced is:

```
(THFIND ALL $?X1 (X1)
  (THGOAL(#IS $?X1 #BLOCK))
  (THFIND 3 $?X2 (X2)
    (THGOAL(#IS $?X2 #PYRAMID))
    (THGOAL(#SUPPORT $?X2 $?X1))))
```

Once this is evaluated, it returns a list of all the blocks satisfying the description, but no record of what pyramids supported them. If the next sentence asked "Are they tall?", we would have no objects for "they" to refer to. Special instructions are inserted into our PLANNER descriptions which cause lists like this to be saved. The actual PLANNER expression produced would be:

```
(THPUTPROP (QUOTE X1)
  (THFIND ALL $?X1 (X1)
    (THGOAL(#IS $?X1 #BLOCK))
    (THPUTPROP (QUOTE X2)
      (THFIND 3 $?X2 (X2)
        (THGOAL(#IS $?X2 #PYRAMID))
        (THGOAL(#SUPPORT $?X2 $?X1)))
      (QUOTE BIND)))
  (QUOTE BIND)))
```

This only occurs when the system is handling discourse.

Finally, "it" can be used in a phrase like "Do it!" to refer to the entire main event of the last sentence. This LASTEVENT is saved, and SMIT can use it to replace the entire meaning of "do it" with the description generated earlier for the event.

When "that" is used in a phrase like "do that", it is handled in a similar way, but with an interesting difference. If we have the sequence "Why did you pick up the ball?" "To

build a stack." "How did you do it?", the phrase "do it" refers to "Pick up a ball". But if we had asked "How did you do that?", it would refer to building a stack. The heuristic is that "that" refers to the event most recently mentioned by anyone, while "it" refers to the event most recently mentioned by the speaker.

In addition to remembering the participants and main event of the previous sentence, the system also remembers those in its own responses so that it can use them when they are called for by pronouns. It also remembers the last time reference, (LASTIME) so the word "then" can refer back to the time of the previous sentence.

Special uses of "it" (as in "It is raining.") are not handled, but could easily be added as further possibilities to the SMIT program.

4.3.2 Substitutes and Incompletes

The next group of things the system needs to interpret involves the use of substitute nouns like "one", and incomplete noun groups like "Buy me two." Here we cannot look back for a particular object, but must look for a description. SMIT looks through a list of particular objects for its meaning. SMONE (the program used for "one") looks back into the input sentence instead, to recover the English description. "One" can be used to stand for part or all of that description.

As with "it", "one" can refer back to something in a previous sentence, the previous reply, or earlier in the same sentence. Here though, there are no restrictions about where in the parsing tree the description can lie. "One" depends more on surface characteristics than on structural differences. For example, it cannot refer back to a NG which is a pronoun or uses a TPRON like "anything". Our program for "one" is not as complex as the one for "it". It is primarily based on the heuristic of "contrast". People often use "one" to contrast two characteristics of basically similar objects, for example "the big red block and the little one." The program must understand these contrasts to interpret the description properly. We realize that "the little one" means "the little red block", not "the little big red block" or "the little block". In order to do this, our system has as part of its semantic knowledge a list of contrasting adjectives. This information is used not only to

decide how much of the description is to be borrowed by "one", but also to decide which description in a sentence "one" is referring to. If we say "The green block supports the big pyramid but not the little one." It is fairly clear that "one" refers to "pyramid". But if we say "The big block supports the green pyramid but not the little one.", then "one" might refer to "block". The only difference is the change of adjectives -- "big" and "little" contrast, but "green" and "little" do not. Our program looks for such contrasts, and if it finds one, it assumes the most recent contrasting description is the referent. If there is no contrast between the phrase being analyzed and any NG in the same sentence, previous answer, or previous sentence, it then looks for the most recent NG which contains a noun.

It is interesting to note that SMONE causes the system to parse some of its own output. In order to use the fragment of a NG it finds, SMONE must know which elements it can use (such as noun, adjective, and classifier) and which it does not (such as number and determiner). For the noun groups in previous inputs, the parsing is available, but for the reply, only the actual words are available and it is necessary to construct a simple parsing before understanding the meaning of "one". It does not call the entire system recursively to do this, but uses a simplified version.

An incomplete NG, containing only a number or quantifier is

used in much the same way as "one". In fact, if we look at the series "Buy me three." "Buy me two." "Buy me one.", we see they are nearly identical. We can take the view that an incomplete NG actually has an implied substitute noun of "one". This is the way our program handles incomplete noun groups.

Currently the set of contrasts is stored separately as special properties in the dictionary entries of the adjectives involved. It would be better to combine this with the semantic marker system, or the actual system of PLANNER programs and concepts.

4.3.3 Overall Discourse Context

We have discussed several ways of using overall discourse context in understanding. We have so far experimented with only one of these -- keeping track of what has been mentioned earlier in the discourse. This is not the same as looking back in the previous sentence for pronoun references, as it may involve objects several sentences back or occurring in separate sentences. If there are many blocks on the table, we can have a conversation: "What is in the box?" "A block and a pyramid." "What is behind it?" "A red block and another box." "What color is the box?" "Green." "Pick up the two blocks."

The phrase "the two blocks" is to be interpreted as a particular pair of blocks, but there may be others in the scene, and nowhere in the dialog were two blocks mentioned together. The system needs a way to keep track of when things were mentioned, in order to interpret "the" as "the most recently mentioned" in cases like this.

To do so, we use PLANNER'S facility for giving properties to assertions. When we mention a "green block", the semantic system builds a PLANNER description which includes the expressions:

```
(THGOAL(#IS $?X1 #BLOCK)) (THGOAL(#COLOR $?X1 'GREEN))
```

After the sentence containing this phrase has been interpreted, the system goes back to the PLANNER descriptions and marks all of the assertions which were used, by putting the current sentence number on their property lists. This is also done for

the assertions used in generating the descriptions of objects in the answer.

When the semantic programs find a definite NG like "the two red blocks", the second NG specialist (SMNG2) uses PLANNER to make a list of all of the objects which fit the description. If there are the right number for the NG, these are listed as the "reference" of the NG, and the interpretation of that NG is finished. If there are fewer than called for by the determiners and numbers, SMNG2 makes a note of the English phrase which was used to build the description, and returns a message to the parser that something has gone wrong.

If the parser manages to parse the sentence differently, all is well. If not, the system assumes that the NG interpretation was the reason for the failure, and the system uses the stored phrase to print out a message "I don't understand what you mean by..."

However, if there are too many objects which match the description, SMNG2 tries to find out which were mentioned most recently. It does this by using PLANNER to recheck the description for the items it found, but this time using only those assertions mentioned in this or the previous sentence. This is easily done by using PLANNER's ability to put a "filter" on the assertions to be retrieved from the data base, allowing only those which are accepted by a LISP predicate (which in this case looks for a sentence number in the appropriate range on the

property list of the assertion. See <Sussman> for details.) If it finds the right number, these must be the reference of the NG. If it finds too few, it can reiterate the procedure, but using all of the assertions mentioned in the last two sentences. This backward progress continues until at some point either it finds the right number or the number found jumps from below the right number to above it. In this case a message of failure is returned to the parser as before, but a marker is set so that in case the sentence cannot be understood, the message returned is "I don't know which... you mean", as in sentence 2 of Section 1.3:

I DON'T UNDERSTAND WHICH PYRAMID YOU MEAN.

Of course, a more sophisticated program is needed for determining the referents of definite NGs. If the conversation is about a book, the NG "the title" is clearly acceptable and must refer to the title of that book, even though no title was previously mentioned. If there is a complicated NG like "the biggest cat in the world" or "the thing I want to do most", the program must deduce from the form of the NG that there is only one such object, even though it has not been previously mentioned, and that it is being defined by this description. There is a large area for expansion in dealing with these and similar problems.

It seems likely that these and other similar aspects of language can be handled systematically, using the formalism of

procedures within a system like the one described in this paper. The semantic and deductive systems provide a framework within which we can formalize and combine a variety of aspects of meaning and knowledge to deal with the real problems of semantics.

4.4 Generation of Responses

In this section we will describe how our language-understanding system generates its linguistic responses. This aspect was not emphasized as much in the research as the problem of understanding language, and we have not developed a general theory of discourse generation. The programs produce behavior which seems reasonable for carrying on a dialog about the robot world. In particular we dealt with four different aspects of discourse production -- patterned responses, appropriate answers to questions, generating descriptions of objects and events, and the production of fluent discourse.

4.4.1 Patterned Responses

The easiest way to get language behavior from a computer program is to include a set of fixed responses which are appropriate for the situations it will encounter. These responses can be made as complex and varied as we want, since they are created by the programmer, and the program only repeats them. Many language systems make heavy use of these fixed responses to achieve "good sounding" output.

In general, these responses are not flexible enough for a true language system, but there are places where they are appropriate -- for example, when the program wants to acknowledge that something has happened, or that it has reached a certain point in its analysis. Our system uses several types of patterned response. First there is a set of simple responses

for specific situations. The system responds "OK." when a command is carried out, "I UNDERSTAND." when a declarative sentence is analyzed, "I DON'T UNDERSTAND." when a sentence cannot be analyzed, and "I CAN'T." when a command cannot be executed.

A slightly more complex type of response involves "filling in the blank" with a phrase borrowed from the input. The simplest example in our system is "SORRY, I DON'T KNOW THE WORD "...", PLEASE TYPE TWO SPACES." The offending word is taken as it was input and inserted in the blank. The "two spaces" are to clear the input buffer of characters typed after the message was sent. Two slightly more complex types of response involve manipulating the determiners of the phrase which was input. If the user types something like "the three green pyramids", and the system cannot figure out what he is referring to, it types "I DON'T KNOW WHICH THREE GREEN PYRAMIDS YOU MEAN." It has simply replaced "the" with "which" before filling the blank. The "I assume" mechanism does the opposite, replacing an indefinite determiner with "the". If we talk about "some green pyramid" or "a green pyramid", then later refer to that pyramid as "it", the system can notify us of its interpretation of "it" by saying "BY 'IT' I ASSUME YOU MEAN THE GREEN PYRAMID." Here the system has removed the indefinite determiner or quantifier ("a" or "some") and replaced it with "the". It uses knowledge about various kinds of determiners to fill in the pattern

reasonably in a number of cases (for example, possessives are left alone, since they are already definite). It can also handle the use of "they" to refer to several objects.

Our most complex "blank-filling" response is the one which handles ambiguity. It uses parts of the input sentence and pieces of the definitions of words stored in the dictionary. In the response to sentence 24 of the sample dialog (section 1.3):

24. how many things are on top of green
cubes?
the system typed:

I'M NOT SURE WHAT YOU MEAN BY "ON TOP
OF "IN THE PHRASE "ON TOP OF GREEN
CUBES " .

DO YOU MEAN:
1 - DIRECTLY ON THE SURFACE
2 - ANYWHERE ON TOP OF ?

The two phrases "on top of" and "on top of green cubes" were taken directly from the input, and the paraphrases "directly on the surface" and "anywhere on top of" were part of the dictionary definition provided for "on". One of the answer routines looks at the list of places where the interpretations differ, and generates this request for clarification. The response must eliminate at least one possibility, and the program continues generating questions like this one until only one interpretation remains. If the ambiguity is caused by multiple interpretations for a pronoun, the English text of the phrases originally used to refer to the objects are used in the response choices.

4.4.2 Answering Questions

In order to carry on a discourse, we need to know what types of responses people expect to different types of questions. In Section 2.3.3 we classified various types of questions syntactically, and we can use this classification to select appropriate answers. The system answers whenever possible with a complete phrase, providing as much information as possible. No attempt is made to produce full sentences, since in fact people rarely answer questions with sentences. The natural response to "Which block is in the box?" is "the red block", or "the red one", certainly not "The red block is in the box." We have tried to analyze what sort of phrases provide the most informative and natural responses.

The first case is the WH- question, and within that class, the NGQ(see section 2.3.2), involving a question NOUN GROUP, such as "which block", "who", "how many pyramids", or "what". We can divide these into three classes -- "how many", "vague", and "specific". If asked "What is in the box?", we can answer "a blue block and a pyramid" without being more specific at identifying them. If instead we are asked "Which block is in the box?" we must use a more specific description like "the large blue cube which supports a pyramid." The program must generate English descriptions of particular objects, and it must be able to generate both definite and indefinite descriptions. This part of the program will be described in the next section.

The use of its results is straightforward for NGQ questions. If the NG is "what", we generate indefinite descriptions of the object or objects. If it is "which...", we generate a definite description. "Who" is never a problem, since the system only knows of two people, "you", and "I". There are also default responses, so that a question like "Which block supports the table?" can be answered "NONE OF THEM."

HOWMANY questions are answered with the number of appropriate objects, followed by "OF THEM" to make the discourse smoother. For example, the response to dialog sentence 6, "How many blocks are not in the box?", is "FOUR OF THEM."

The next type of question is the QADJ, such as "why", "when", "how", or "where". The only three which have been implemented so far are "why", "when", and "how", but the others can be done in an analogous fashion. A "why" question is answered using the system's memory of the subgoals it used in achieving its goals in manipulating toy objects. If we can decide what event is being referred to in the question, we can see what goal called it as a subgoal, and answer by describing the higher goal in English. If the event was itself a top level goal, it must have been requested as a command, and the response is "BECAUSE YOU ASKED ME TO."

We need a program which creates an English description of an event from its PLANNER description like (#PUTON :B3 :TABLE). It must generate phrases which refer to the objects involved,

and combine them into a clause of the proper type with the proper tense. This program is also described in the next section.

We can use the same event-describer to answer "how" questions by describing all of the events which were subgoals used in achieving the event mentioned. We say "BY...", then list each event in an "ing" form, as in: "BY PICKING UP A RED BLOCK AND PUTTING IT IN THE BOX." If the event was itself a lowest-level goal, the system has no way of analyzing its own programs for achieving that goal, and answers "I CAN'T ANALYZE HOW".

"When" questions are answered similarly -- a time is named by describing the top-level goal which was being carried out at the time, saying "WHILE I WAS..." and using the "ing" form to describe the event. This is inappropriate if the question refers directly to the top-level goal (We can't answer "When did you build the stack?" with "WHILE I WAS BUILDING THE STACK."), and in that case we say "BEFORE..." and name the top-level goal immediately following in time. If the goal mentioned was just done, the system replies "JUST NOW.". In addition to the normal responses, the system has a set of fixed responses such as "NEVER", and "I CAN'T EXPLAIN A NON-EXISTENT EVENT." to answer questions which demand them.

Finally we come to YES-NO questions which, paradoxically, are the most complicated. It seems that a one word answer is

called for, but this is often impossible and rarely the best way to respond. If we ask "Does the block support three pyramids?", and in fact it supports four, what is the correct answer? The system could ask for clarification of the implicit ambiguity between "at least three" and "exactly three", then answer "yes" or "no". But it is more efficient and helpful to answer "FOUR OF THEM", leaving the speaker to interpret his own question. If there were only two pyramids, the system could respond "NO", but it would be more informative to say "NO, ONLY TWO OF THEM". In any of these cases, we might be even more complete, and say something like "FOUR OF THEM: A BLUE ONE, TWO RED ONES, AND A LARGE GREEN ONE."

There are three parts we can include in the answer -- "yes" or "no", a number, and a description of the objects. Remember that whenever possible a YES-NO question is treated like a NG question by looking for a focus element (an indefinite NG in the question). A question like "Does the pyramid support a block?" is treated logically like the question "Which block does the pyramid support?", or "How many blocks does the pyramid support?" All three send a THFIND ALL request to PLANNER, asking for all of the objects fitting the description:

```
(THPROG (X)
  (THGOAL (#IS $?X #BLOCK))
  (THGOAL (#SUPPORT :B5 $?X)))
```

where :B5 is the system's internal name for the pyramid being referred to by "the pyramid" (this would be determined in

separate calls to PLANNER). In the case of the HOWMANY question we answer with the number of objects found. For the "which" question, we name the objects. In the case of a YES-NO question, we answer with all three kinds of information, saying "YES, TWO OF THEM: A LARGE RED ONE AND THE GREEN ONE."

The first element is "YES" if the answer is clearly yes (for example if the number is matched exactly, or the number in the original question was indefinite as in this example), "NO" if it is clearly no (for example if there are fewer than requested, none at all, or the request was of a form "exactly..." "at least..." "more than..." etc. and was not met), and is omitted if there is a question about its interpretation (as described above).

The second element, the number, is omitted if the number found matches the request (For example, "Are there three blocks?" is not answered redundantly, "YES, THREE OF THEM: A GREEN ONE AND TWO LARGE RED ONES."). The phrase "OF THEM" following the number is changed to "THINGS" if the focus contains a TPRON like "anything", or "something". If the number found is less than that in the focus, it is preceded by "ONLY...", so the answer comes out "NO, ONLY TWO OF THEM:...")

At the end of a response, we put the description of the objects found, unless the request used a special number format such as "exactly...", "at least..." etc. In which case the system assumes the number is more important than the specific

objects. We use the object-naming program in its indefinite mode. If the focus originally appeared as the object of a preposition, we repeat that preposition before the description to clarify the answer. Thus, "Is the pyramid on a block?" is answered "YES, ON A LARGE GREEN ONE." The unknown agent of a passive like "Is it supported?" is implicitly the object of "by", so the answer is "YES, BY THE TABLE." If a YES-NO question contains no possible focus since all of its NGs are DEFINITE, as in "Does the table support the box?", the system answers simply "YES" or "NO".

4.4.3 Naming Objects and Events

The previous section covers all of the different types of questions the system can handle, and the types of phrases it uses in response. We have not yet explained how it names an object or describes an event. This is done with a set of PLANNER and LISP functions which examine the data base and find relevant information about objects. These programs take advantage of the fact that the subject matter is limited. In general, the way an object is named is highly dependent on what the person being spoken to is interested in and what he already knows. This has not been dealt with yet. Certain features of objects, such as their color and size, are assumed to be the best way to describe them in all contexts.

First we need to know how the object is basically classified. In the BLOCKS world, the concept #IS represents this, as in (#IS :HAND #HAND), (#IS :B1 #BLOCK), and (#IS #BLUE #COLOR). The naming program for objects first checks for the unique objects in its world, "I", "you", "the table", "the box", and "the hand". If the object is one of these, these names are used. Next it checks to see if it is a color or shape, in which case the English name is simply the concept name without the "#". The question "What shape is the pyramid?" is answered "POINTED." since it has the shape #POINTED. If the object is not one of these and is not a #BLOCK, #BALL, or a #PYRAMID the program gives up. If it is one of those three, the correct

noun is used (including a special check of dimensions to see if a #BLOCK is a "cube"), and a description is built of its color and size. At each stage of building, the description is checked to see if it refers uniquely to the object being described. If so, the determiner "the" is put on, and the description used without further addition. If there is only one ball in the scene, it will always be referred to as "the ball".

If the description includes color and size, but still fits more than the desired object, the outcome depends on whether a specific description or an nonspecific one is called for. If it is nonspecific, the program puts the indefinite pronoun "a" or "an" on the beginning and produces something like "A LARGE GREEN CUBE". If it is specific, more information is needed. If the object supports anything, the program adds the phrase "WHICH SUPPORTS..." then includes the English descriptions (indefinite) of all the objects it supports. If the object supports nothing, the program adds "WHICH IS TO THE RIGHT OF..." and names all of the objects to the left of the desired one. This still may not characterize the object uniquely in some situations, but the system assumes that it does. If at any point in the dialog, an object is given a proper name, it is referred to using only the noun and the phrase "NAMED...", as in "THE BLOCK NAMED SUPERBLOCK."

Naming events is relatively straightforward. With each event type (such as #PUTON or #STACKUP) we associate a small

program which generates an English name for the event and combines it properly with the names of the objects involved. For example, the definition for #PUTON is:

```
(APPEND (VBFIX (QUOTE PUT)) OBJ1 (QUOTE (ON)) OBJ2)
```

VBFIX is a program which puts the verb into the right form for the kind of clause needed to answer the question. (for example, -ing for answering "how", or infinitive for answering "why"). It takes into account the changes in spelling involved in adding endings. OBJ1 and OBJ2 are bound by the system to the English names of the objects involved in the event, using the object-naming program described above. APPEND is the LISP function which puts together the four ingredients end to end. We therefore get descriptions like "BY PUTTING A LARGE RED CUBE ON THE TABLE". There is a special check for the order of particles and objects, so that we output "TO PICK UP THE SMALL BLUE PYRAMID.", but, "TO PICK IT UP" rather than, "TO PICK UP IT".

4.4.4 Generating Discourse

The previous sections described a generating capability which can produce reasonable English answers to different types of questions, but used by themselves, the features described would produce awkward and stilted responses which would at times be incomprehensible. Even though we have mentioned some discourse-like patterns (like "...OF THEM" following a number), we have not yet discussed the real problems of discourse. The system uses three different discourse devices in producing its answers. These are much more limited than the range of discourse features it can understand, but they are sufficient to produce fluent dialog.

The first problem involves lists of objects. Our initial way of naming more than one object is to simply string the descriptions together with commas and "AND". We might end up with an answer like "YES, FOUR OF THEM: A LARGE BLUE BLOCK, A SMALL RED CUBE, A SMALL RED CUBE, AND A SMALL RED CUBE." To avoid this redundancy, the object-namer looks for identical descriptions and combines them with the appropriate number to get "A LARGE BLUE BLOCK AND THREE SMALL RED CUBES." (Note that it also must change the noun to plural).

The next problem is the use of substitute nouns. We would like to respond to "Is there a red cube which supports a pyramid?" by "YES, A LARGE ONE." Instead of "YES, A LARGE RED CUBE." By comparing the English descriptions of the objects with

the wording of the focus in the input sentence, we can omit those nouns and adjectives they share and replace them by "ONE".

The third problem is more serious, as ignoring it can lead to incomprehensible responses. Consider the answer to question 32 in the dialog ("How did you do it?"). If we did not use the pronoun "it" or the determiner "that", the response would be:

BY PUTTING A LARGE RED BLOCK ON THE TABLE, THEN LETTING GO OF A LARGE RED BLOCK, THEN PUTTING A LARGE GREEN CUBE ON A LARGE RED BLOCK, THEN LETTING GO OF A LARGE GREEN CUBE, THEN PUTTING THE RED CUBE ON A LARGE GREEN CUBE, THEN LETTING GO OF THE RED CUBE.

How many different blocks and cubes are involved? In describing events, we must have some way to indicate that we are referring to the same object more than once. We can do this using "it" and "that", and at the same time can use these words to improve the smoothness of the discourse in other ways. The system has heuristics which lead it to use "it" to refer to an object in an event it is describing whenever:

1. the same object was called "it" in the question.
2. the object was called "it" in the question preceding the current one, and "it" was not used in the current one.
3. the object has already been named in the current answer, and was the first object in the answer
4. no objects have yet been named in the current answer, and the object was the only one named in the previous answer.

To refer to an object already named in the current answer, other than the first, the program applies the determiner "that" to the appropriate noun, to get a phrase like "BY PUTTING A

GREEN BLOCK ON A RED CUBE THEN PUTTING THAT CUBE IN THE BOX."

4.4.5 Future Development

The generation of language is a complex subject, wide open to future development. Our current system is just a beginning, and has some major deficiencies. First, we would like to describe an object by using facts which are relevant to the context. In our simple world, we have declared by fiat that color, size, and support relationships are the important facts about an object. We could just have well have used location to get answers like "the block nearest to the back of the table". With a wider range of subjects, we would need much more sophisticated heuristics for deciding what features of an object will serve best to identify it to the hearer.

Second, we do not have a way to turn an arbitrary PLANNER expression into English. We can handle only specific objects and simple events. There are a number of applications for a more powerful English generator. For example, in case of ambiguity, we shouldn't have to include special paraphrases in the definition. The system should be able to look at the two PLANNER descriptions and describe the difference directly in English.

The system should be able to tell us more about itself and how it does things. If we ask a question like "How do you build stacks?", it should be able to look at its own programs and convert them to an English description like "First I find a space, then I choose blocks, then I put one of the blocks on

that space, then..." PLANNER's structure of goals and subgoals is ideal as a subject for this kind of description, and a great deal could be done along this line. In a more speculative vein, the development of discourse generators which could convert an internal logical format into natural language might lead to computer essay writers, or translators which could understand the material they were working with.

4.5 Comparison with Other Semantic Systems

4.5.1 Introduction

This section compares our semantic system with two other models for the semantics of natural language. Each has served as the basis for computer programs which "understand" language, and we will consider the usefulness of the models for this purpose as well as their value as theoretical models of a natural process.

The three models can be labelled "categorization", "association", and "procedure". These do not represent a cross section of semantic theories, but represent one particular type of theory. They assume that it is meaningful to postulate a conceptual organization of human knowledge, related by semantics to the linguistic forms used in expressing thoughts. This sets them off from traditional approaches which avoided postulating mentalistic structures and dealt instead with extra-mental representations such as logical truth conditions or stimulus-response relationships.

These three models are oriented towards viewing language as a human activity rather than an abstract calculus of symbols. They study the process in terms of human models, and take into account the production and interpretation of language. One possible reason why the current syntactic theories have been weak in developing semantic theories is their insistence on a "neutral" characterization of the competence of a language user,

without regard to the process carried out in an intelligent speaker or hearer. Since semantics involves the interaction between the structures of the language and the knowledge and intelligence of the language user, it cannot be understood without trying to deal with this intelligence directly.

4.5.2 Categorization

The Categorization model of semantics was developed in its best known form by Katz and Fodor (Katz), and has been a part of many computer systems for understanding natural language.

The basic principle is a structure of categories, called "semantic markers", dividing the conceptual world much as the Dewey decimal classification subdivides the books in a library. The usual sense of the word "bachelor" has the semantic markers "male", "human", "animate", "physical object", etc. and the final distinguishing characteristic ("never having been married") is its "semantic distinguisher".

In choosing between different senses of a word in a particular sentence, these markers are combined according to "projection rules". For example, the word "colorful" would be interpreted in one sense in "colorful cube", another in "colorful party", while the rules would indicate that "a colorful ball" has two possible readings.

Further information can be gleaned from the logical relations between the markers such as the fact that "male uncle" is redundant, while "female uncle" is anomalous.

Fodor and Katz did not attempt to explain the process of producing and understanding language in terms of these markers, preferring to see them as abstract neutral relationships underlying the speaker's competence. They did not deal in any systematic way with those aspects of meaning which cannot be

dealt with through this type of categorization.

Nevertheless, categorization has been used in many computer programs for understanding natural language, to help choose the right meaning for potentially ambiguous words. Schank (Schank 1969, 1970) has extended the application of this theory, using semantic relationships to parse sentences conceptually.

Associated with each sense of a word is a conceptualization, specifying the semantic relationships of that word with other words in the same structure. For example, one meaning of the word "hit" would be an action of physical striking, whose subject is a "person", whose object is a "physical object", and which has a possible instrument of the category "weapon". The sentence "I hit the boy with a stick." would be parsed by noticing words in categories which could fill the roles, and by setting up an appropriate structure. It could also account for the interpretation in which "hit" involves striking with a fist, and "with" represents possession of a "physical object" by a "person", but this would be found only on "prompting" (see (Schank 1970), p. 26).

The underlying belief is that humans make much use of this sort of categorization in understanding sentences, rather than doing a complete syntactic parsing. The sentence:

"The window the ball the boy threw hit broke."

is understood more easily than:

"The man the woman the girl knew liked died."

Fodor and Garrett <Fodor 1967> studied sentences like these, and found that they are more easily understood when the categories associated with the verbs can indicate the conceptual structure. For sentence fragments and ungrammatical utterances, this ability seems vital.

4.5.3 Association

The second model has its roots more in psychology, and the presence of "associations" between words in a person's mind. It postulates a sprawling collection of words and concepts, connected to each other by simple links. If "goose" is connected to "quill" and "quill" to "pen" and "pen" to "ink", there is a path of length three from "goose" to "ink". These links would be present in information such as "A quill is a goose feather", "Pens can be made of quills", etc. The justification for this model is that the course of a path describes the relationship between the two nodes, and that its length is a measure of their relatedness. The use of association as a model for computer language understanding has been most influenced by the work of Quillian (Quillian 1964, 1969). Information is coded into the network of concepts using several types of links (for example, the class-subclass relationship used in the categorization model). In understanding a sentence, a search is initiated through the network from each of the content words of the sentence, to find the shortest paths linking them. The system uses the information along that path to decide what the sentence is about.

It is important to understand why I call this the "association" model instead of the "network" model. The word "network" has been used to refer to every conceivable variety of data structure. The semantic markers in the categorization model form a network, Schank (Schank 1969) refers to the output of his parser as a "language-free conceptual network",

recent parsers <Woods 1970> are called "augmented transition networks", while our parser uses systemic grammar with "system networks". Each of these "networks" represents a completely different structure and use of data. Saying that a structure is a "network" is not much more informative than saying that it is represented by bits in a computer memory. What must be stated is the way the network is used.

A central commitment of the association model is that there is a significance to tracing along the links from node to node ignoring their content. Once a path is found, all sorts of logical operations may be used to determine its significance and make use of its information, but in the propagation, a minimum of calculation is done at each node.

It is difficult to formalize a "minimum" of calculation, but it is important to have some understanding of its implications. Any computation whatever, can be expressed as a network by drawing a flow chart, with the blocks of computation as nodes, and the transfers of control as links. The computation then traces a path through the net. It might seem that there is something inherently different between a program following a single path through a flowchart, and a signal propagating in all directions through a net. However the difference disappears if we allow some sort of parallel processing (for example the pseudo-simultaneous evaluation of several paths, as found in many simulation languages, and some theorem-provers such as new versions of PLANNER <Newitt 1970>). This is not the place to debate the merits of parallel vs. serial processing. The important thing is to realize that once

networks carry out computations at each node, they move away from the association model, towards a model of semantics as a program.

Some types of natural language understanding do appear to involve simple associations. On hearing the words "fortune" and "almond", a person will (if he has eaten in Chinese restaurants) think of the word "cookie". It is hard to describe logical connections which lead to such a quick reaction, and much more appealing to picture a short association path between "cookie" and each of the original words. The model can also be used to explain the choice of a single meaning for a potentially ambiguous word in a sentence. If a word is connected by a link to each of the concepts it might describe, the shortest path to the other words in the sentence should be through the relevant meaning.

4.5.4 Procedure

We can call the model of semantics used in our system the "procedure model". The primary organization of knowledge is in a deductive program with the power to combine information about the parsing of the sentence, the dictionary meanings of its words, and non-linguistic facts about the subject being discussed. Any relevant bit of knowledge can itself be in the form of a program or procedure to be activated at an appropriate time in the process of understanding. The program operates on a sentence to produce a representation of its meaning in some internal language, in our case PLANNER. This language allows the expression of a wide variety of the aspects of language -- logical connectives and quantifiers, time references (provided by verb tenses and modifiers), different sorts of object-modifier relationships, types of object reference (e.g. the difference between "the dog" and "a dog"), etc.

In analyzing a sentence, the program can use information about previous sentences in the discourse and about the subject being discussed. This allows it to deal with features of language such as pronoun reference, substitute nouns, the effect of discourse on specific referents, and the disambiguation of meaning through knowledge of non-linguistic facts (like Bar-Hillel's classic example of the "box in the pen" <Bar-Hillel>).

Other programs, such as <Woods 1969> also use the procedure model. These programs use a complete syntactic parsing of the

Input to provide a framework from which the program can decide what aspects of meaning to deduce. The majority of the language-comprehending programs have used the procedure model in a simplified form, performing only a few elementary types of deduction in analysis, and having an internal language tailored to a specific application.

4.5.5 Evaluating the Models

There is no single set of criteria to judge the success of a model of semantics or a computer program for "comprehending language". Success is relative to the goals of the model, and the aspects it wishes to describe. Four criteria seem to be of importance both in computer language comprehension and in developing a theory of semantics. These are: ability to combine syntax and meaning ; efficiency; ability to explain human performance; and the ability to understand language in context. These will be discussed separately.

A. Integrating Syntax

There are many facets to the meaning of an utterance in a natural language, and no model sheds equal light on all of them. In fact, two of the three models are limited to one part of the meaning -- the basic semantic relationships between the words used in the sentence.

The Fodor-Katz version of the categorization model does not attempt to deal with the part of meaning expressed by the semantic distinguishers, analyzing only those aspects which can be modelled by the markers. It does not work with other aspects of meaning such as tense, mood, and reference to objects. Schank's version attempts to model the way people understand a sentence, describing an actual parsing process. However the conceptual parsing does not actually find the "meaning". One argument for the model is human ability to understand utterances

in a foreign language without a detailed knowledge of the grammar. Only the meanings of words are needed to find the language-free conceptual connections (see <Schank 1970>, p.4). This seems a good parallel to the type of understanding done by the categorization model. Anyone who has tried to get along in an unfamiliar foreign language will be familiar with the following experience. A friend says something which contains the foreign language equivalents of the words "like", "see", and "film". The foreign visitor knows the words and their "word-concept couplings", but is totally at a loss in trying to respond, since the sentence may have been any one of a vast set, including:

"I like seeing films." "Have you seen any films you liked?" "I see you like films." "Would you like to see a film?" "I saw a film I liked." "Did you like seeing the film?" etc.

Without the additional meaning provided by syntax, it is impossible to understand the content of the sentence. If the visitor responds "We are talking about a person who sees a film and likes the film", his foreign friend can rightfully reply "Oh, you didn't understand." This problem applies equally to the association model. Finding the intersection of signals from the nodes "see", "film", and "like" might produce the right conceptual relationships, but none of the additional information. Neither of these two models has been the basis for an actual question-answering system, since they do not deal with the ways in which syntax conveys meaning, and therefore ignore

those aspects of semantics in which syntax plays a large role. The procedure model pays more attention to a complete understanding of a sentence, trying to interweave syntax, semantics, and deduction in order to actually answer a question, use a piece of new information, or follow a command.

B. Efficiency

It would seem reasonable to look for capabilities of the other two models not possible for the procedure model. In one sense, this quest is a joke. Since a procedure system has information in the form of programs, those programs can include simulations of any other model. The significant question is not what is possible theoretically, but what is reasonable to do.

A program could play a simple game like NIM by using standard strategies of minimax and look-ahead. If it could win, it would provide a successful model of NIM playing. However it makes no use of the simple winning strategy, and therefore is a bad model for the specific game. Similarly, the procedure model approaches semantics in a general way, saying that every part of semantics involves powers of deduction and the ability to combine information of a variety of types. If in fact, major parts of language comprehension can be explained by more elementary approaches, the general procedure model is not a good description for those areas.

The issue at stake is more than computer efficiency. Since we are modelling a natural process, the criterion of "Occam's

"razor" applies just as in any other science. The most satisfactory explanation is the least complex one which can account for the facts.

The area of computational complexity is barely charted, and it is nearly impossible to determine that some computation is inherently "more complex" or "more difficult" than another. It is especially dangerous to characterize high-level processes like deduction, since a computation which takes impossibly long using one scheme may be trivial for another. However, there is an intuitive sense in which efficiency can be judged. A procedure system could handle the "fortune -- almond" example by systematically looking through the things it knows about fortunes and almonds, and using some sort of analogy program to test for relationships. This seems clearly more complex than the presence of a simple association link. Those who advocate the associational model feel that there will be many such cases in which the deductive process needed to find the path would be impossibly torturous and lengthy.

There is also a complexity of syntactic parsing. The semantic connections might give clues to the underlying structure which would change the parsing task into simply checking the plausibility of the relations, and cleaning up the details. This is the approach taken by both Schank and Quillian. The example involving the boy, ball, and window involves a complex syntactic structure which could not be

handled by most of the parsers which have been written for computers, yet a simple set of semantic criteria seem to analyze it directly without any complex syntax rules.

C. Modelling Human Behavior

It is difficult to find psychological experiments which could decide between one model and another, since the underlying conceptual structure is too complex to isolate a "single interaction". However, some examples seem to suggest the validity of various models. In speech communication, people understand sentence fragments, scattered words, and blurred phrases which require filling in much of the meaning. In a model requiring a complete parsing, this would add a great deal of complexity, since the parser would have to know about the different types of fragments as well as the grammatical sentences. A semantic relation model suggests that the syntax is only used at the end of the process, to check on the conceptual message. If the syntax is lacking, the final check is gone, but the basic meaning is still discoverable.

Special types of language use, like poetry, puns, and jokes seem to involve simple associational links. Often the punch line of the joke comes from recognizing the inappropriateness of the link which was made, while the poem conveys meaning by showing that a link is not as irrelevant as it outwardly seems, but hints at deeper connections.

D. Context

One of the most important facts about language comprehension is that sentences do not appear in logical isolation, but are always part of a context, both of other utterances and of the situation in which they are uttered. Some semanticists try to avoid this problem, saying it goes beyond the proper realm of semantics. Katz and Fodor believe that "...a semantic theory cannot be expected to account for the way setting determines how an utterance is understood." (<Katz> p. 486) However, if semantics is to be a study of the way language and meaning are actually related, we cannot ignore the facts.

One of the main strengths of the procedure model is its ability to include all sorts of knowledge in making deductions at any stage of semantic analysis. The program can call on the contextual knowledge just as easily as the dictionary definitions or syntax. Within the framework of the basic procedure model, there can be a detailed model of those parts of the context which are needed for understanding (for example a memory of the objects which have been mentioned, so pronouns can refer back to them). The examples below show some of the problems involved in other models when context enters into understanding.

Schank uses the sentence "I hit the man with a stick." to illustrate conceptual parsing. (<Schank 1970> p. 26) Since "hit" takes an instrumental of the type "weapon", the conceptual

parser first assumes that the phrase "with a stick" has this meaning. Let us look at two possible contexts for the sentence:

1. Three men attacked me. I hit the man with a stick.
2. A man attacked me. I hit the man with a stick.

The phrase "with a stick" is interpreted differently in the two cases, but this cannot be because of differing conceptual relations. The relevant information is that a phrase like "the man" will only be used when it is clear which particular man is meant, while "the man with a stick" will be used only in trying to distinguish one particular man from others.

Another example used (<Schank 1970> p. 11) is the disambiguation of the word "fly" depending on whether its subject is a "pilot" or not. If you know that Ed's father is a pilot, the sentence "Ed's father flew to Chicago." should be interpreted in the sense of "operating a plane". But there are no categorization clues in the words "Ed" or "father", for the conceptual parser.

It seems that within the association model there should be some way to make use of this information. If there were a node linked to "Ed", "father", and "pilot", then the network search involving "Ed", "father", and "fly" would go through it, and the relations could be determined from the path. This approach is deceptive, as adding this type of knowledge creates a world of false short paths. This is a problem inherent to the approach.

The earlier example of "goose" "quill" "pen" "ink" could be extended one place further from "ink" to "spot". It is extremely unlikely that a sentence containing "spot" and "goose" would actually involve this connection. Yet it might be a much shorter path than the one actually representing the connection in "I spotted a goose." As the amount of knowledge grows, there will be a rapidly expanding number of false paths, made up of links which are individually very close, but which bear no logical relation to each other. Since the association network does not check the logical relations of the links until after the path is found, there is no immediate way to sort these out.

By including specific knowledge, this problem is exacerbated, since each node will have a large number of links, of widely differing logical types.

There are various ways to sneak deduction into the association model, and for each simple example, it is possible to design a trick which cuts out the irrelevant links (Quillian's distinction between property and superset links <Quillian 1969> is an example). As the amount of information in the net increases, it needs more ad-hoc deductive schemes, and in order to handle language generally, the association net will become a full-fledged parallel processor using procedures to find semantic relations.

In addition to swamping the system with implausible links, the association model can produce very plausible incorrect

links. If the sentence:

"He hated the landlord so much he decided to move into the house on Harvard St."

were given to a system like TLC, it would be hard to restrain it from saying "We are talking about a landlord who owns a house. The house is located on Harvard St...." The path from "landlord" to "house" will be as strong as the path from "lawyer" to "client" in the standard associational example. But in this case, deduction is needed to realize that the association is wrong. A person would move out of the house of a landlord he hated, not into it.

These examples point out a serious defect of non-deductive models. Earlier sections discussed the existence of areas of language comprehension which could not be handled without syntax and deduction. These examples indicate that deduction is necessary even for the tasks for which the other models are designed -- finding the semantic and conceptual relations between the words in the sentence.

4.5.6 Conclusions

Although the procedure model seems to account for much more of language behavior than either of the others, there are some parts of language understanding where they are especially applicable. There is no one aspect of understanding which can succeed without deduction, but the simpler models can be of help in making appropriate deductions in many sentences.

In trying to understand language at a deep level, a system cannot hope to simply throw together the advantages of these different models, but needs a way to integrate them usefully. This is the primary advantage of the procedure point of view. It is flexible enough to make use of the other models in a systematic way in an integrated system. If semantic criteria can simplify parsing, a partial semantic analysis can be included in the procedure before syntactic analysis. This could involve category matching, or even a controlled search through some sort of association net. These might be used as well in choosing between meanings of a word, or in finding information applicable to a deduction about the subject matter in generating a response.

The degree to which these special sub-models could be used would depend on the particular application and the subject being discussed.

A program to read poetry or retrieve documents on the basis of vague descriptions of subject matter may need a strong

association component. One written to answer questions about airline schedules, or understand questions and commands to a robot will be more procedurally oriented. In general, a use involving detailed knowledge of any specific subject can rely on deduction, while an application needing superficial knowledge of a wide range of subjects can benefit from association, and will be correspondingly weak in its ability to give specific responses.

Specific models such as association and categorization are subparts of a model of language understanding, while the procedure model is fundamentally an approach to integrating all of the different sub-models into a total semantic theory. It represents a point of view that no part of the process can be isolated from the basic computational power or "intelligence" of the language-understander (whether human or computer). Understanding of language, as well as other types of human behavior, depends on this ability to see the operation of intelligence at every level of processing.

Chapter 5. Conclusions

5.1 Teaching, Telling and Learning

One of the most important requirements of a natural language understanding system is generality. It should not be based on special tricks or shortcuts which limit it to one particular subject or a small subset of grammar, but should be expandable to really handle the full diversity of language. In each of the three preceding chapters we have pointed out that many approaches to language understanding are quite limited, and have tried to illustrate the progression within each sub-area towards more general approaches.

5.1.1 Types of Knowledge

In evaluating the flexibility of a system, we must consider the four different levels of knowledge it contains.

First, there is the "hard core" which cannot really be changed without remaking the entire system. This is its "innate capacity" -- the embodiment of its theory of language. At this level we must deal with such questions as whether we should use a top-down transformational parser, a semantic net, or some other approach to the basic analysis of a sentence, or whether we should have special tables of information or a general notation (such as the predicate calculus) for representing information.

The second level of knowledge is the complex knowledge about the language and the subject being discussed. This would include such things as the grammar of a language, or the conceptual categories into which the speaker divides his model of the world. If we think about the human speaker, this is a type of knowledge which is obviously not innate (since the grammar would be different for English and Chinese and the set of concepts used would differ for talking about toy blocks and talking about love stories). However it is not something which he learns by being told, or which he changes very easily. Over a period of years, he builds up a store of very complex, interrelated knowledge, which serves as a framework for more specific information.

The third level is our storehouse of knowledge about the details of our language and our world. It includes the meanings of words, and most of what we called "complex knowledge" in section 3.1.3. This would include such things as "A house built on sand cannot stand.", "If you want to pick up a block, first clear off its top.", or "Sunspots cause strange weather.". In human terms, this is the knowledge we are continually learning all of our lives, and forms the bulk of what we are taught in school.

Finally, the fourth level is the set of specific facts which are relevant to a discussion. This includes facts such as "Flight 342 leaves Boston at noon.", "The red block is 3 inches tall.", or "A banana is hanging above the chair.". This is the easiest type to learn, since it does not demand forming any new interrelationships. It is more like putting a new entry into a table or a new simple assertion into a data base. There is no sharp distinction between levels three and four, but within any given system there will usually be two different ways of handling information corresponding to this distinction. Let us look at the three areas of syntax, inference, and semantics, and see how these different levels of knowledge relate to language understanding programs and the way they can learn.

5.1.2 Syntax

In syntax it is clear that at the top level of knowledge there will be a basic approach to grammar, whether it be transformations, pattern matching, or finite state networks. In addition, there must be some sort of built in system to carry out the parsing.

Some programs (such as the early translation programs) had the grammar built in as an integral part of the system. In order to add new syntactic information it was necessary to dig into the deepest innards of the system and to understand its details. It was recognized quite early that this approach made them inflexible and extremely difficult to change. The majority of language systems have instead adopted the use of a "syntax-directed" parser. A grammar is described by a series of rules which are applied by a uniform parsing procedure. In handling simple subsets of English, this turns grammar into a third-level type of knowledge. We can add new single rules (for example, adding the fact that verbs can have a modifying adverb) in a way similar to adding words to a vocabulary -- without worrying about the interaction between rules. This simplicity is deceptive, since it depends on the simplicity of context-free grammars for small subsets of natural language. Once we try to account for the complexities of an entire language with something like a systemic or transformational grammar, we must again pay attention to the complex interrelationships between

the rules, and the grammar becomes a tangled web into which any new addition must be carefully fitted. For examples of the complexity of current transformational grammars for English, see <Klima>. More recent programs which use transformational grammars <Woods 1970>, <Bobrow 1970> <Thorne 1968, 1969> recognize the fact that syntax is not really that simple, and adopt a more interrelated representation such as networks.

In our system we have used programs to express the grammar, as explained in chapter 2. This is not a return to the original first-level representation, since the grammar programs are completely separate from the system itself. One of the arguments for using syntax-directed parsers was that the grammar rules could be expressed in a uniform way which did not depend on the details of the parsing program. Therefore changes could be made more easily and the grammar was expandable. By designing a special language for writing grammars, we can use a representation which is just as general as syntax-rule tables, but which allows greater flexibility in designing a grammar, and relating it to semantics.

How difficult is it to change our grammar? For small changes (like allowing noun groups to contain only a number, as in "Are there any books? I want three.") only one or two additional lines of program would be needed. For a more substantial change (like adding a new type of modifying clause) we might need as many as a dozen small additions to the grammar

in different places which would be affected. The first change could be done with little difficulty by anyone with an understanding of PROGRAMMAR and section 2.3. The second would take a deeper understanding of how the grammar is written, but would still involve only a small amount of programming, and of course would not involve changing the basic system at all. The grammar was written to be fairly complete and with expansion in mind. It seems flexible enough that we will be able to include as much of the the complexity of English as we want.

What is important in terms of learning is that this is level-two knowledge -- it is the type of knowledge which is learned once in a lifetime by a person (or computer program), and should not need any major changes after childhood. Therefore although it must be changeable, we do not need to worry about "quick" learning techniques. If any learning is studied at this level, we must deal in a sophisticated way with the methods used to learn large amounts of complex interrelated material. Those computer programs which have "learned" syntax (<McConlogue> <Siklossy>) have done so by taking such an oversimplified view of syntax that the results hardly have significance for natural language.

At level three of our knowledge of syntax, we have our knowledge of particular words, their grammatical categories and peculiarities. We need to ask, How easy is it to add new words? How much do we have to know about the grammar to increase the

vocabulary? In most systems there are a few words (such as "be", "there", or "than") which have complex and unique grammatical behavior. These are built into the grammar initially at level two. The rest of the vocabulary, like nouns and verbs, can be specified in a simple format. Our system is no exception. To add the words "cat", "purple", and "walk" to the system, we would only need to know the right abbreviations (from section 2.3) to enter in LISP:

```
(DEFLIST WORD (CAT (NOUN NS))(PURPLE(ADJ))(WALK(VB INF ITRNS)))
```

This says that "cat" is a singular (NS) NOUN, "purple" is an ADjective, and "walk" is the INFinite form of an InTRaNSitive VerB.

Can we give this information in English? It would be straightforward to add the right terms to the vocabulary and set up simple PLANNER theorems which would allow us to say "'Cat' is a noun." or "'walk' is an intransitive verb." It would be an interesting project to see how far this could be extended. Some programs have avoided giving dictionary entries to these "open class" words (like verbs, nouns, and adjectives) and let the parser determine their part of speech from context. (Thorne 1969) This approach is not generally meaningful for a complete language understanding system, since we need a dictionary of meanings. It could be used when adding new words to the system, and could be done so trivially in our input programs, by assigning all unknown words to have all possible "open class"

grammatical features, then letting the parser choose the correct ones for the context.

5.1.3 Inference

In the domain of Inference, there has been tremendous variation in how different systems treat knowledge. In the early programs, all of the complex information was at level one (built into the system), while the specific facts were at level four. As we have discussed, this made it very hard to modify or expand the complex information held by the system. In the theorem provers, all of the complex information was treated at the fourth level -- as a set of individual formulas which were treated as isolated facts. At level one, they have a uniform proof procedure as the heart of the system. We have discussed how this lack of information at other levels (information about the interrelationships between different theorems) severely limits this approach. In our system, only simple assertions (such as "Noah is the father of Jafeth.", or "Parent-of is the converse of Child-of.") are dealt with at the lowest level. The rest of the knowledge is in the form of PLANNER theorems which have the ability to include information about their connections to other theorems. Some of these, such as the examples in section 3.1.3 about canaries and thesis evaluation, are at the third level, since they are not interwoven into complex relationships with other parts of the knowledge. Other theorems, such as the BLOCKS programs (section 3.4) for keeping track of a table full of objects, are at level two.

Again we can ask, how easy is it to add or change

Information at each of the levels. At the two ends, the answer is clear. At the top we have PLANNER and our commitment to its kind of theorem-proving procedures. Any change in this is a major overhaul. At the bottom level, we have simple facts like "The red pyramid is supported by the green cube." These are the facts which the system plays with whenever it is conversing. They can be changed by simply telling information (either in English or PLANNER), and are changed automatically when things happen in the world (for example if we move the red pyramid). The middle levels form the much more interesting problem.

At the second level we have our basic conceptual model of the world. This includes our choice of categories for objects, ways of representing actions, time, place, etc. One of the benefits of PLANNER (and of LISP, in which it is embedded) is that we have a variety of useful facilities to represent our world efficiently. Section 3.4 described the BLOCKS world, and it should be similarly easy to define new worlds of discourse for the system (see below for examples).

The third level presents the most interesting problems for adding new information to the system. It is simple to do so in PLANNER by adding new theorems, but we would like to do it in English as well. Of the previous systems, the only ones which could accept complex information in English were the theorem provers which dealt with it at the fourth level (as a set of unrelated formulas). In our sample dialog, we have some

examples of telling the system simple and slightly complex information in English. Saying "I like blocks which are not red, but I don't like anything which supports a pyramid." created two theorems. The first says, "If you want to prove I like something, prove that it is a block and that it is not red." This is no different from a formula for any theorem prover, since it is not related to the system in any complex way. The second theorem says, "If you are trying to prove that I like something, and you can prove that it supports a pyramid, then give up." This interacts with the other goals and theorems, but in a very specialized way.

Much smarter programs could be built to accept complex information and use it to actually modify the PLANNER theorems already in the data base. For example, we might have a theorem to pick up a block, but it fails whenever the block has something on top of it. We would like to say in English, "When you want to pick up a block, first take everything off of it.", and have the system add this information to the theorem in the form of an additional goal statement at the beginning. In order to do this, the system must have not only a model of the world it talks about, but also a model of its own behavior, so that it can treat its own programs as data to be manipulated and modified. This is one of the most fascinating directions in which the system could be expanded.

Another is the possibility of letting the system learn from

experience. This is a complex problem and can be dealt with at many levels. At a simplistic level, we can have it "learn" specific facts. For example, we have a theorem which proves that a block has its top clear (by proving it supports nothing). As the last line of this, we have the PLANNER statement (THASSERT (#CLEARTOP \$?X)), which says that we should add to the data base the assertion that this block is clear. If we then need the fact again, we don't need to repeat the deduction. In a sense the system has "learned" this fact, since it has been added to the data base without being mentioned in the dialog. But in another sense, it hasn't learned any new information, since nothing can be deduced with this fact that couldn't have been done before using the theorem that already existed. A more interesting type of learning would be shown by changing the PLANNER theorems for accomplishing a goal, depending on what had been achieved in the past. For example, we might have a goal statement with the recommendation (THTBF THTRUE) meaning try anything you can. If the goal is achieved using some particular theorem, we might have the system change the recommendation to suggest trying that theorem first. At a more advanced stage, we would have a heuristic program which tried to figure out why a particular chain of deduction worked or didn't work in a particular case. It would then modify the recommendations to choose the best theorems in whatever environments came up in the future. It might also recognize the need for new theorems in

some cases, and actually build them. This is perhaps closest to human learning. It does not involve juggling parameters or adding new isolated bits of information. Instead it involves figuring out "How are my ideas wrong (or right)?" and "How can I change or generalize them?" It involves a kind of "debugging" of ideas, and is a key reason for representing knowledge as procedures.

5.1.4 Semantics

Since semantics is the least understood part of language understanding, it is difficult to find a clear body of "level one" knowledge on which to base a system. Our system has a basic approach to semantics, explained in chapter 4, but most of the semantic work is done at level two -- the interrelated group of LISP programs for handling particular semantic jobs. At this level we have two separate areas of knowledge. The first is knowledge about the language, and the way it is structured to convey meaning. This includes knowledge such as "In a passive sentence, the syntactic subject is the semantic object.", "A definite noun group refers to a particular object in the world model." or "'It' is more likely to refer to the subject of the previous sentence than the object." This is closely tied to the grammar, and is about as hard to modify as the grammar programs themselves. The other type of level two knowledge is the network of "semantic features" described in section 4.2. This is peculiar to the domain being discussed, and becomes more complex as the range of discussion increases. As we pointed out, this is currently separate from the network of "concepts" used for inference by PLANNER, but the two could be combined. As with level two knowledge in other areas, this is not something to be quickly learned and changed. Our knowledge of how language conveys meaning grows along with our knowledge of its syntactic structure, and is just as seldom modified.

At the third level we have the bulk of semantic information -- the meanings of individual words. This is the part which must be easy to change and expand. As in most language understanding systems, this knowledge is in the form of separate dictionary entries, so that new words can be added without changing others. The definition of each word is a program in the "semantic language" described in section 4.2, and we gain great flexibility from this program form. The writer of semantic definitions does not have to be concerned with the exact form of the grammar, and if he wants to enter simple words, he can use a standard function to describe them very simply. Most words can be added by using the functions CMEANS and NMEANS, or by using the particular simple semantic form appropriate to the type of word (for example, we would define "thirteen" by ((NUM 13))). If we come across a type of semantic problem or relationship we hadn't anticipated, or which involves relating things in an unusual way, we can write a LISP function as the definition of the word to perform the required operations.

We have tried to design our system so that it would be flexible and could be easily adapted to handle other fields of knowledge and to have a large vocabulary. It would be nice to enter new definitions in English instead of having to use the special semantics language. In our sample dialog, the sentence "A "steeple" is a stack which contains two green cubes and a

pyramid." produced a new definition for a noun. This is only possible when we can express the definition of the new word in terms of the old words and concepts. It is a bit deceptive for a language understanding system to allow new words to be added so simply. If we wanted to define the word "face", so that we could talk about the faces of blocks, the system would be lacking the basic concepts and relationships necessary to use the new word. This kind of knowledge is at the second level, and we cannot expect to add it through a simple definition. There must be a powerful heuristic program which recognizes the need for a new concept and which relates this concept to the entire model of the world. In this example, it would have to realize that a face is a part of an object, but is not an object itself. This might have varied consequences throughout the model, wherever relations such as "part" are involved.

Thus although our system can accept definitions of some words, it is a worthwhile but untried research project to design programs which will really be able to learn new words in an interesting way. We believe that this will be much easier within the environment of a problem solving language like PLANNER, and that such programs could well be added to our system.

5.2 Directions for Future Research

In the preface we talked about using computers in a new way. We speculated about the day when we will just tell our computer what we want done, and it will understand. This paper has described a small step in that direction. Where is such research leading? What approaches should we take in the future?

We can see three basic directions in which we could extend our system. First, at present it knows only about a tiny simplified subject. Second, most of what it knows has to be programmed, rather than told or taught. Finally, we can't talk to it at all! We have to type our side of the conversation and read the computer's.

The problem of widening the scope of knowledge involves much more than building bigger memories or more efficient lookup methods. If we want the computer to have a large body of knowledge, the information must be highly structured. The critical issue is to understand the kinds of organization needed. One of the reasons that our system is able to handle many aspects of language which were not possible in earlier systems is that it has a deep understanding of the subject it is discussing. There is a whole body of theorems and concepts associated with the words in the vocabulary, and by making use of this knowledge in its question-answering and action, its language behavior is more like ours. In going to larger areas of discourse we cannot give up this insistence that the computer

must know what it is talking about.

We need a way to integrate large amounts of heterogeneous knowledge into a single system which can make use of it. At the same time, we cannot let the system become overburdened and inefficient by insisting on a stifling generality and uniformity. We want the advantages of specialized types of knowledge and structure that can come from limiting the subject to a small area, but at the same time we must have the flexibility that allows knowledge of different types to interact. PLANNER-like languages may be a beginning toward these new kinds of organization.

There are many different approaches which can be taken towards higher organization of knowledge. We may want to think in terms of a "block-structure" of contexts, each of which carries its own special vocabulary and information. We may think of a network, in which we can consider the "distance" between two concepts or words. It might be possible to deal with a set of specialized "subroutines" for dealing with different kinds of situations. Even for something as seemingly simple as childrens' stories, there are tremendous complexities and a well-structured approach is necessary.

In section 4.1.4 we discussed some of the ways our system could take advantage of this large-scale structure of knowledge. The subject matter would influence the choice of relevant definitions of words and appropriate theorems to be used in

dedication. This has been explored very little, and there are many possibilities for further research.

The problem of learning is of great interest not only to those working on practical computer systems, but also to psychologists interested in understanding how learning takes place in other intelligent systems, such as people. We need to understand how the amount of knowledge we already have affects the amount and the way we can learn. Working on a natural language program offers several advantages for studying problems of knowledge and learning. Language represents a body of highly complex knowledge, which itself can provide a rich field for learning tasks with a wide range of difficulties. Also, language is a major vehicle through which people learn about the world. In studying the way that a computer could accept new information in natural language, we are studying a key area in learning. We need to understand the ways in which learning depends on the organization of our knowledge. We need to explore in what ways knowing about its own mentality could allow a computer to really learn. This is perhaps the most interesting possibility for research, and we have discussed it at length in Section 5.1.

We have discussed the difficulties involved in accepting new declarative knowledge in any but a superficial way. One of the problems most closely associated with this is the use of world-knowledge in understanding declarative sentences. Compare the

sentences:

I put the heavy book on the table and it broke.

I put the butterfly wing on the table and it broke.

The understanding of the referent of the pronoun "it" must depend on the likelihood of the different objects breaking or causing breakage. This could be handled by having the declarative sentence "Interpreter" try out both interpretations and see which leads to more "reasonable" conclusions.

Our system can currently do this only if the knowledge of the world needed is a specific simple fact ("there is no block in the box.") or a categorical fact ("table can't pick up blocks.") A more complex system is needed to accept general declarative statements and explore their consequences. It must seek the interpretation which is neither trivial nor incongruous, but which provides new information as the speaker must have intended it to. Contextual factors play the major role. The expectations might be completely reversed if the sentence were preceded by "The strangest thing just happened!"

Finally we have the problem of speech communication with computers. Again the issue is not one of more efficient hardware, but one of knowledge. Spoken language calls on the listener to fill in a great deal from his own knowledge and understanding. Words, phrases and whole ideas are conveyed by fragments and mumbles which often serve as little more than a clue as to what they intend. The need for a truly vertical

system is much greater for speech than for written language. The analysis at even the lowest level depends on whether the result "makes sense." People can communicate under conditions where it is nearly impossible to pick out individual words or sounds without reference to meaning.

In our system we tried to integrate the syntactic, semantic and deductive programs in a flexible way. We allow meaning to guide the direction of the parsing. Our semantic interpretation is guided by logical deduction and a rudimentary model of what the speaker knows. For spoken language this must be expanded. Perhaps we might look for fragments of sentences and use their meaning to help piece together the rest. Or possibly we could create a unified system in which the deductive portion could look at the context and propose what it thought the speaker might be saying, on the basis of meaning, and the audible clues in the utterance. It might be possible to have a more multi-dimensional analysis in which prosodic features such as voice intonation could be used to recognize important features of the utterance. This is not at all saying that we should throw syntax overboard in favor of some sort of vague relational structure. Often the most important clues about what is being said are the syntactic clues. What is needed is a grammar which can look for and analyze the different types of important patterns rather than getting tremendously involved with finding the exact details of structure in a fixed order. Systemic

grammar is a step in this direction, and the use of programs for grammars gives the kind of flexibility which would be needed for doing this kind of analysis. It is not clear whether our system in its present form could be adapted to handle spoken language, but its general structure and the basic principles of its operation might well be used.

The challenge of programming a computer to use language is really the challenge of producing intelligence. Thought and language are so closely interwoven that the future of our research in natural language and computers will be neither a study of linguistic principles, nor a study of "artificial" intelligence, but rather an inquiry into the nature of intelligence itself.

Appendix A - Index of Syntactic Features

Underlines indicate primary description of feature.

-OB 148	FINITE 137
-OB2 148	FUTURE 133, 134
ACTV 116, 119, 137, 138	IMPER 137
ADJ 120, 121, 132, <u>139</u>	IMPERATIVE 104, 105, 106
ADJG 121, <u>131-132</u> , 139	INCOM 123, 125, 142
ADJQ 105, 107	INDEF 123, 124, 125, 142
ADJREL 105	ING 105, 109-113, 137, 138, 148
ADJUNCT 105, 109, 128-131	INGOB2 148
ADV <u>139-141</u>	INGQ 105
ADVADV 141	INGREL 105
ADVMEASQ 105, 107	INT 116, 118, 148
ADVMEASREL 105	IT 116, 119
AGENT 116, 119, 128, 129	ITRNS 116, 117, 148
AND 150, 151	ITRNSL 116, 118, 148
AS 131	ITSUBJ 105, 113, 119
AUX 146	LIST 150
BE 116, 137	LISTA 150
BINDER <u>141</u>	LOBJ 105, 117, 128, 129
BOTH 150	LOBJQ 105
BOUND 105, 109	LOBJREL 105
BUTNOT 150	MAJOR 105, 106, 107
CLASF 120, 121, 124, <u>141</u>	MASS 142, 143
CLAUSE <u>104-119</u> , 121, 143, 148	MEASQ 105, 106
CLAUSEADV 141	MEASREL 105, 111
COMP 117-118, 123, 126-128, 131	MODAL 133, 134, 148
COMPAR 121, 131, 132, 139	NDET 123, 124
COMPONENT 151	NEED2 145
COMPOUND <u>149-152</u>	NEG 123, 125, 137, 138, 142
COMPQ 105, 108	NFS 123, 127, 145
COMPREL 105, 111, 113	NG <u>120-127</u> , 128-129, 132
DANGLING 105-107, 111, 113, 129	NGQ 107
DECLARATIVE 105, 106	NOBJ 123
DEF 123, 124, 142	NONUM 142, 143
DEFPOSS 123, 127	NOR 150
DEM 123, 124, 142, 145	NOUN 120, <u>143</u>
DET 120, 122-124, <u>141-143</u>	NPL 123, 127, 142-145
DOWNQ 105, 109	NS 123, 127, 142-145
DOWNREL 105, 111	NUM 120, <u>144</u>
DPRT 116, 119	NUMD 123, <u>144</u>
EN 105, 110, 137, 138, 148	NUMDALONE 144

NUMDAN 144	QADJ <u>146</u>
NUMDAS 144	QAUX <u>146</u>
NUMDAT 144	QDET 142
NUMDET 125	QNTFR 123, 124, 125, 142
OBJ 105, 126, 145	QUEST 123, 125, 128-132, 145
OBJ1 105, 112, 123	QUESTION 105-107, 146, 146
OBJ1Q 105, 108	RELADJ 111, 113
OBJ1REL 105, 111	REDEL 105, 111
OBJ1UPREL 113	RELPREPG 128, 129
OBJ2 105, 112, 123	REPOB 148
OBJ2Q 105, 108	REPORT 105, 112, 148
OBJ2REL 105, 111	RSNG 105, 109-113, 119, 128, 148
OBJQ 105	RSQ 105, 109, 121
OBJREL 105	SEC 105, 106
OF 123, 125, 128, 129	SHORT 105, 107, 129
OFD 142	SHORTREL 111, 113
OFD-INCOM 142	SUBING 105, 112, 148
OFOBJ 123, 126	SUBJ 105, 112, 117, 123, 126, 145
OR 150	SUBJFORK 151
ORD 120, 122, <u>144</u>	SUBJQ 105, 107
PAST 133, 134, 148	SUBJREL 105, 111
PASV 116, 119, 135, 137, 138	SUBJT 116, 123
POSES 123, 125, 126	SUBJTQ 105, 108
POSS 123, 126, 143, 145	SUBJTREL 105, 111, 113
POSSDEF 145	SUBQ 105
PREP 128, <u>144</u> , 145	SUBREL 105
PREP2 145	SUBTO 105, 112, 148
PREPADV 141	SUBTOB 148
PREPG 121, <u>128-130</u>	SUP 121, 144
PREPOBJ 105, 112, 123, 126, 128	THAN 131
PREPQ 105, 106, 107	THAT 105, 112
PREPREL 105, 110, 111, 113, 129	THERE 108, 116
PRESENT 133, 134	TIMI 143
PRON <u>145</u>	TIME 123, 143
PRONG 123, 124, 126, 127	TIMEQ 105, 108
PRONREL 145	TIMEREL 105, 111
PROPN 124, <u>145</u>	TO 105, 109-113, 137, 138, 148
PROPNG 123, 124	TODEL 137, 138
PRT 116, 118, 119, <u>145</u>	TPRON 124, 132, <u>146</u>
Q 120, 128, 129, 131	TPRONG 123, 124

TRANS 116, 117, 148
TRANS2 116, 117, 148
TRANS2TOQ 105
TRANS2TOREL 105
TRANSL 116, 117, 148
TRANSTO2Q 108
TRANSTO2REL 111
UPQUEST 128, 129
UPREL 105, 113, 128, 129
V3PS 148
VB 146, 148
VBAD 141
VBFORK 151
VFS 148
VG 133-138, 148
VPRT 148
WH- 105, 106, 107
WHRS 105, 110, 113
WORD 139-148
Yes-No 105, 106

Appendix B - Sample Parsings

HOW MANY EGGS WOULD YOU HAVE BEEN GOING TO USE IN THE CAKE IF
YOU HADN'T LEARNED YOUR MOTHER'S RECIPE WAS WRONG?

```
((HOW MANY EGGS WOULD YOU HAVE BEEN GOING TO USE IN THE
  CAKE IF YOU HADN'T LEARNED YOUR MOTHER'S RECIPE WAS
  WRONG)
(CLAUSE MAJOR QUEST NGQUES POLR2 ACTV OBJ1Q TRANS)
(((HOW MANY EGGS)
  (NG QUEST HOWMANY NDEF NPL DET)
  ((HOW (QDET)) (MANY (QDET)) (EGGS (NOUN NPL))))
  (WOULD (VB AUX MODAL QAUX))
  ((YOU) (NG SUBJ DEF NS NPL) ((YOU (PRON NPL NS SUBJ OBJ))))
  ((HAVE BEEN GOING TO USE) (VG MODAL NAGR (FUT PAST MODAL))
    ((WOULD (VB AUX MODAL QAUX))
      (HAVE (HAVE VB AUX INF TRANS))
      (BEEN (AUX VB BE EN))
      (GOING (VB ITNS ING))
      (TO (TO))
      (USE (VB INF TRANS MVB))))
  ((IN THE CAKE) (PREPG)
    ((IN (PLACE PREP PLACE))
      ((THE CAKE)
        (NG OBJ DET NS DEF)
        ((THE (DET NPL NS DEF)) (CAKE (NOUN NS))))))
  ((IF YOU HADN'T LEARNED YOUR MOTHER'S RECIPE WAS WRONG)
    (CLAUSE BOUND DECLAR ACTV TRANS)
    ((IF (BINDER))
      ((YOU) (NG SUBJ DEF NS NPL) ((YOU (PRON NPL NS SUBJ OBJ))))
      ((HADN'T LEARNED)
        (VG VPL V3PS NEG (PAST PAST))
        ((HADN'T (HAVE VB AUX TRANS PAST VPL V3PS VFS NEG))
          (LEARNED (VB TRANS REPOB PAST EN MVB))))
      ((YOUR MOTHER'S RECIPE WAS WRONG)
        (CLAUSE RSNQ REPORT OBJ OBJ1 DECLAR BE INT)
        (((YOUR MOTHER'S RECIPE)
          (NG SUBJ NS DEF DET POSES)
          (((YOUR MOTHER'S)
            (NG SUBJ NS DEF DET POSES POSS)
            (((YOUR) (NG SUBJ POSS)
              ((YOUR (PRON NPL NS SUBJ OBJ POSS)))
              (MOTHER'S (NOUN NS POSS))))
            (RECIPE (NOUN NS))))
          ((WAS) (VG V3PS VFS (PAST))
            ((WAS (AUX VB BE V3PS VFS PAST MVB))))
          ((WRONG) (ADJG Q COMP) ((WRONG (ADJ))))))))))
```

PICK UP ANYTHING GREEN, AT LEAST THREE OF THE BLOCKS, AND
EITHER A BOX OR A SPHERE WHICH IS BIGGER THAN ANY BRICK ON THE
TABLE.

```
((PICK UP ANYTHING GREEN /, AT LEAST THREE OF THE BLOCKS /, AND
  EITHER A BOX OR A SPHERE WHICH IS BIGGER THAN ANY BRICK ON
  THE TABLE)
(CLAUSE MAJOR IMPER ACTV TRANS)
(((PICK) (VG IMPER) ((PICK (VPRT VG INF TRANS MVB))))
  (UP (PRT))
  ((ANYTHING GREEN /, AT LEAST THREE OF THE BLOCKS /, AND
    EITHER A BOX OR A SPHERE WHICH IS BIGGER THAN ANY
    BRICK ON THE TABLE)
  (NG OBJ OBJ1 EITHER COMPOUND LIST NS)
  (((ANYTHING GREEN) (NG OBJ OBJ1 TPRON)
    ((ANYTHING (NS TPRON)) (GREEN (ADJ))))
  ((AT LEAST THREE OF THE BLOCKS)
    (NG OBJ OBJ1 COMPONENT NUMD NUM NPL DET OF)
    ((AT (AT))
      (LEAST (NUMD NUMDAT))
      (THREE (NUM))
      ((OF THE BLOCKS)
        (PREPG OF)
        ((OF (PREP))
          ((THE BLOCKS)
            (NG OBJ DET NPL DEF)
            ((THE (DET NPL NS DEF)) (BLOCKS (NOUN NPL))))))
    ((A BOX OR A SPHERE WHICH IS BIGGER THAN ANY BRICK ON THE
      TABLE)
    (NG OBJ OBJ1 COMPONENT OR COMPOUND BOTH NS)
    (((A BOX) (NG OBJ OBJ1 COMPONENT DET NS INDEF)
      ((A (DET NS INDEF)) (BOX (NOUN NS))))
    ((A SPHERE WHICH IS BIGGER THAN ANY BRICK ON THE TABLE)
      (NG OBJ OBJ1 COMPONENT DET NS INDEF)
      ((A (DET NS INDEF))
        (SPHERE (NOUN NS))
        ((WHICH IS BIGGER THAN ANY BRICK ON THE TABLE)
          (CLAUSE RSQ SUBREL BE INT)
          (((WHICH) (NG RELWD DEF NPL) ((WHICH (NPL))))
            ((IS) (VG V3PS (PRES))
              ((IS (AUX VB BE V3PS PRES MVB))))
          ((BIGGER THAN ANY BRICK ON THE TABLE)
            (ADJG Q COMP COMPAR THAN)
            ((BIGGER (ADJ COMPAR))
              (THAN (THAN))
              ((ANY BRICK ON THE TABLE)
                (NG SUBJ COMPAR DET NS QNTFR)
                ((ANY (DET NS NPL QNTFR))
                  (BRICK (NOUN NS))
                  ((ON THE TABLE)
```

(PREPG Q)
((ON (PREP PLACE))
((THE TABLE)
(NG OBJ DET NS DEF)
((THE (DET NPL NS DEF))
(TABLE (NOUN NS))))))))))

Appendix C - Sample BLOCKS Theorems

```

(DEF THEOREM TA-CONTAIN
  (THANTE (X Y Z)
    (#AT $X ?)
    (THGOAL (#MANIP $X))
    (THGOAL (#SUPPORT $Y $X))
    (THOR (THAND (THGOAL (#IS $Y #BOX))
      (THVSETQ $Z $Y))
      (THGOAL (#CONTAIN $Z $Y)))
    (THASSERT (#CONTAIN $Z $X))))

(DEF THEOREM TC-MAKESPACE
  (THCONSE (SURF SIZE OBJ SPACE X (WHY (EV)) EV)
    (#FINDSPACE $SURF $SIZE $OBJ $SPACE)
    (THNOT (THGOAL (#IS $SURF #BOX)))
    (MEMORY)
    TAG
    (THAND (THGOAL (#SUPPORT $SURF $X))
      (THGOAL (#GET-RID-OF $X)
        (THUSE TC-GET-RID-OF)))
    (THOR (THGOAL (#FINDSPACE $SURF
      $SIZE
      $OBJ
      $SPACE)
      (THUSE TC-FINDSPACE))
      (THGO TAG))
    (MEMOREND (#MAKESPACE $EV $SURF)))

(DEF THEOREM TC-MORE
  (THCONSE (MEASURE X Y)
    (#MORE $MEASURE $X $Y)
    (THVSETQ $MEASURE
      (GET $MEASURE (QUOTE MEASFN)))
    (GREATERP ($MEASURE $X)
      ($MEASURE $Y)))

(DEF THEOREM TC-ON
  (THCONSE (X Y Z)
    (#ON $X $Y)
    (THOR (THGOAL (#SUPPORT $Y $X))
      (THAND (THASVAL $X)
        (THGOAL (#SUPPORT $Z $X))
        (THGOAL (#ON $Z $Y)
          (THUSE TC-ON))))))

```

```

(DEFTHEOREM TC-PICKUP
  (THCONSE (X (WHY (EV)) EV)
    (#PICKUP $?X)
    (MEMORY)
    (THGOAL (#GRASP $?X) (THUSE TC-GRASP))
    (THGOAL (#RAISEHAND)
      (THNODB)
      (THUSE TC-RAISEHAND))
    (MEMOREND (#PICKUP $?EV $?X))))

(DEFTHEOREM TCT-PICKUP
  (THCONSE (X EV TIME)
    (#PICKUP $?X $?TIME)
    (THOR (THAND (THGOAL (#PICKUP$?EV $?X))
      (TIMECHK $?EV $?TIME))
      (THGOAL (#PICKUP $?EV $?X $?TIME)
        (THUSE TCTE-PICKUP)))))

(DEFTHEOREM TCTE-PICKUP
  (THCONSE (X EV EVENT TIME)
    (#PICKUP $?EV $?X $?TIME)
    (THOR (THAND (THGOAL (#PICKUP $?EV $?X))
      (TIMECHK $?EV $?TIME)))
    (THSUCCEED))
  (THAMONG $?EVENT EVENTLIST)
  (MEMQ (GET $?EVENT (QUOTE TYPE))
    (QUOTE (#PUTON #GET-RID-OF)))
  (TIMECHK $?EVENT $?TIME)
  (THOR (THGOAL (#PUTON $?EVENT $?X ?))
    (THGOAL (#GET-RID-OF $?EVENT $?X)))
  (THVSETQ $_EV (MAKESYM (QUOTE E)))
  (AND (PUTPROP $?EV
    (PUTPROP $?EV
      (GET $?EVENT (QUOTE END))
      (QUOTE START))
    (QUOTE END))
    (PUTPROP $?EV (QUOTE #PICKUP) (QUOTE TYPE))
    (PUTPROP $?EV $?EVENT (QUOTE WHY))
    (SETQ EVENTLIST (CONS $?EV EVENTLIST))
    (THASSERT (#PICKUP $?EV $?X))))

(DEFTHEOREM TE-CONTAIN (THERASING (X Y)
  (#AT $?X ?)
  (THGOAL (#CONTAIN $_Y $?X))
  (THERASE (#CONTAIN $?Y $?X))))

```

Appendix D - Sample PROGRAMMAR Program
Excerpt from Noun Group Parser

```

DET1
(COND ((ISQ H NS) (FQ NS)) (T (FQ NPL)))
(OR NN (AND (FQ NUMBER) (GO INCOM)))
NUMBER
(FQ DET)
((NQ OF) OF ADJ)
QNUM
((ISQ H NONUM) OF NIL)
((AND (PARSE NUM) (FQ NUM)) NIL OF)
((COND ((EQ (SM H) 1) (AND (CQ NS) (RQ NPL)))
        ((CQ NPL) (RQ NS))))
NIL
(NUMD)
INCOM)
((EQ (CADDR (NB H)) (Q NO)) ADJ NIL)
OF
((AND (NQ OF) (PARSE PREPG OF)) SMOF NIL)
((EQ (CADDR (NB H)) (Q NONE)) INCOM ADJ)
SMOF
(FQ OF)
((OR SMN (SMNGOF) (NOT (POP))) RETSM INCOM)
ADJ
((PARSE ADJ) ADJ CLASF NIL)
EPR
((OR (ISQ H SUP) (ISQ H COMPAR)) NIL REDUC)
(FQ ADJ)
(AND (EQ (CADDR N) (Q OF))
      (PARSE PREPG OF)
      (OR SMN (SMNGOF) (AND (ERT NOUN SMNGOF3) (GO FAIL)))
      (FQ OF)
      (GO RETSM))
(GO INCOM)
CLASF
((OR (PARSE VB ING (CLASF)) (PARSE VB EN (CLASF)) (PARSE CLASF))
  CLASF
  NIL
  REDUC)
NOUN
((PARSE NOUN) NIL RED2)
((AND (CQ TIME) (NOT (ISQ H TIM1))) RED1 NIL)
(SETQ T1 FE)
(COND ((AND (ISQ H MASS) (OR (CQ PART) (NOT (CQ DET))))
      (FQ MASS)))
(COND ((NOT (ISQ H NPL)) (RQ NPL PART)))
(COND ((NOT (ISQ H NS)) (RQ NS)))
(COND ((AND (NOT (CQ DET)) (NOT (CQ NUMD))) (* H)
      (TRNSF NPL MASS)))

```

```

((MEET FE (Q (NS NPL PART MASS))) NIL REDG)
SMNG
((OR (NOT (CQ OFOBJ)) (CQ DET)) NIL FAIL)
(OR SMN (SMNG1) (AND (ERT NOUN SMNG1) (GO FAIL)))
((NOT (ISQ H POSS)) NIL POSS RETSM)
((AND (NEXTW TO)
      (MEET FE (Q (COMP SUBJ)))
      (PARSE CLAUSE RSQ TC)
      (OR SMN (SMRSQ (SM H) SM) (GO POPRET))))
RETSM
NIL)
((AND (OR (NEXTW AS) (NQ COMPAR))
      (PARSE ADJG THANNEED)
      (OR SMN
        (AND (NULL N)
              (CQ SUBJ)
              (ISQ (* C PV) AUX)
              (ISQ PT BE)
              (GO POPRET))
        (SMPREP (QUOTE ADJ))
        (GO POPRET))))
RETSM
NIL)

```

Appendix E - Sample Dictionary Entries

Syntactic Definitions

```
(deflist word
  (LITTLE (ADJ))
  (LONG (ADJ))
  (MAKE (VB INF TRANS))
  (MOVE (VB INF TRANS))
  (NAME (NOUN NS VB INF TRANS))
  (NARROW (ADJ))
  (NICE (ADJ))
  (OBJECT (NOUN NS))
  (OFF (PRT))
  (OUT (PRT))
  (PICK (VPRT VB INF TRANS))
  (PUT (INF PAST VB TRANSL VPRT))
  (POINTED (ADJ))
  (PYRAMID (NOUN NS))
  (RED (ADJ))
  (RELEASE (VB TRANS INF))
  (RIGHT (NOUN NS))
  (ROUND (ADJ)))

(DEFLIST WORD1
  (BEGAN (BEGIN (PAST) (INF)))
  (GAVE (GIVE (PAST) (INF)))
  (SAW (SEE (PAST) (INF)))
  (TOLD (TELL (PAST) (INF)))
```

Semantic Definitions

```

(DEFLIST SMNTC
  (A ((DET T)))
  (ABOVE ((PREP ((T (#LOC #ABOVE T))))))
  (AFTER ((BINDER (END NIL))))
  (ALL ((DET (COND ((CQ OF) (Q ALL))
                    ((MEET (Q (NUM DEF)) FE) (Q DEF))
                    ((Q NDET))))))
  (BALL ((NOUN (NMEANS ((#MANIP #ROUND)
                        ((#IS *** #BALL))))))
  (BIG ((MEASURE ((#SIZE (#PHYSOB) T))
                (ADJ (NMEANS ((#PHYSOB #BIG)
                              ((#MORE #SIZE *** (200 200
                                                    200))))))))
  (BLACK ((ADJ (#COLOR #BLACK)))
  (BLOCK ((NOUN (NMEANS ((#MANIP #RECTANGULAR)
                        ((#IS *** #BLOCK))))))
  (BLUE ((ADJ (#COLOR #BLUE)))
  (BY ((PREP ((T (CMEANS (((#PHYSOB)) ((#PHYSOB))
                            (#NEXTO #1 #2 *TIME)
                            NIL))))))
  (COLOR ((NOUN (NMEANS ((#COLOR) ((#IS *** #COLOR))))))
  (CONTAIN ((VB ((TRANS (CMEANS (((#BOX)) ((#PHYSOB))
                                   (#CONTAIN #1 #2 *TIME)
                                   NIL)
                                   (((#CONSTRUCT))
                                   ((#THING))
                                   (#PART #2 #1 *TIME)
                                   NIL))))))
  (CUBE ((NOUN (NMEANS ((#MANIP #RECTANGULAR)
                        ((#IS *** #BLOCK)
                        (#EQDIM ***))))))
  (EVERYTHING ((TPRON (QUOTE ALL)))
  (FEWER ((NUMD (LIST (Q <) NUM)))
  (FOUR ((NUM 4)))
  (FRIEND ((NOUN (NMEANS ((#PERSON)
                          ((#IS *** #PERSON))))))
  (GRAB ((VB ((TRANS (#GRASP))))
  (GRASP ((VB ((TRANS (#GRASP))))
  (I ((PRON (SETQ SM (Q (FRIEND))))))
  (IT ((PRON (SMIT (Q IT))))
  (NICE ((ADJ (NMEANS ((#THING)
                      ((#LIKE :FRIEND ***))))))
  (NOW ((ADV (OR (EQ (CADR (ASSQ (QUOTE TIME) FE))
                  (QUOTE :NOW))
              (ERT NOW DEFINITION))))

```

```

(THE ((DET T)))
(HEY ((PRON (SMIT (Q HEY)))))
(THICK ((ADJ (NMEANS ((#PHYSOB) ((#MORE #THICKNESS
                                     ***
                                     (0 200 0)))))
        (MEASURE ((#THICKNESS (#PHYSOB) T)))))
(THIN ((ADJ (NMEANS ((#PHYSOB) ((#MORE #THICKNESS
                                     (0 200 0)
                                     ***)))))
        (MEASURE ((#THICKNESS (#PHYSOB) NIL)))))
(THING ((NOUN (NMEANS ((#THING #VAGUE #PHYSOB)
                        ((#IS *** ?)))))
        (WHO ((PRONREL (SETQ SM (QUOTE (ANIMATE))))
              (PRON (SETQ SM (QUOTE (ANIMATE)))))
        (WHY ((QADJ (FQ WHY)))))

```

Appendix F - PLANNER Data for Dialog in Section 1.3

```

(#IS :B1 #BLOCK)
(#IS :B2 #PYRAMID)
(#IS :B3 #BLOCK)
(#IS :B4 #PYRAMID)
(#IS :B5 #PYRAMID)
(#IS :B6 #BLOCK)
(#IS :B7 #BLOCK)
(#IS :B10 #BLOCK)
(#IS #RED #COLOR)
(#IS #BLUE #COLOR)
(#IS #GREEN #COLOR)
(#IS #WHITE #COLOR)
(#IS #BLACK #COLOR)
(#IS #RECTANGULAR #SHAPE)
(#IS #ROUND #SHAPE)
(#IS #POINTED #SHAPE)
(#IS :SHRDLU #ROBOT)
(#IS :FRIEND #PERSON)
(#IS :HAND #HAND)
(#AT :B1 (100 100 0))
(#AT :B2 (100 100 100))
(#AT :B3 (400 0 0))
(#AT :B4 (640 640 1))
(#AT :B5 (500 100 200))
(#AT :B6 (0 100 0))
(#AT :B7 (0 240 300))
(#AT :B10 (300 640 0))
(#SUPPORT :B1 :B2)
(#SUPPORT :B3 :B5)
(#SUPPORT :B6 :B7)
(#CLEARTOP :B2)
(#CLEARTOP :B4)
(#CLEARTOP :B5)
(#CLEARTOP :B7)
(#CLEARTOP :B10)
(#MANIP :B1)
(#MANIP :B2)
(#MANIP :B3)
(#MANIP :B4)
(#MANIP :B5)
(#MANIP :B6)
(#MANIP :B7)
(#MANIP :B10)
(#SUPPORT :TABLE :B1)
(#SUPPORT :TABLE :B3)
(#SUPPORT :BOX :B4)
(#SUPPORT :TABLE :B10)
(#SUPPORT :TABLE :B6)
(#SUPPORT :TABLE :BOX)
(#AT :BOX (600 600 0))
(#IS :BOX #BOX)
(#IS :TABLE #TABLE)
(#CONTAIN :BOX :B4)
(#SHAPE :B1 #RECTANGULAR)
(#SHAPE :B3 #RECTANGULAR)
(#SHAPE :B2 #POINTED)
(#SHAPE :B4 #POINTED)
(#SHAPE :B5 #POINTED)
(#SHAPE :B6 #RECTANGULAR)
(#SHAPE :B7 #RECTANGULAR)
(#SHAPE :B10 #RECTANGULAR)
(#COLOR :B1 #RED)
(#COLOR :B2 #GREEN)
(#COLOR :B3 #GREEN)
(#COLOR :B4 #BLUE)
(#COLOR :B5 #RED)
(#COLOR :B6 #RED)
(#COLOR :B7 #GREEN)
(#COLOR :B10 #BLUE)
(#COLOR :BOX #WHITE)
(#COLOR :TABLE #BLACK)
(#CALL :SHRDLU SHRDLU)
(#CALL :FRIEND YOU)

```

Some of the data is entered initially. The rest can be deduced and asserted by simple antecedent theorems.

BIBLIOGRAPHY

1. <Bar-Hillel 1964> Bar-Hillel, Jehoshua, LANGUAGE AND INFORMATION, Addison Wesley, 1964.
2. <Black 1964> Black, F., "A Deductive Question Answering System," in Minsky (ed.) SEMANTIC INFORMATION PROCESSING, pp. 354-402.
3. <Bobrow 1964> Bobrow, Daniel G., "Natural Language Input for a Computer Problem Solving System," in Minsky (ed.), SEMANTIC INFORMATION PROCESSING, pp. 135-215.
4. <Bobrow 1967> Bobrow, Daniel, "Syntactic Theory in Computer Implementations," in Borko (ed.), AUTOMATED LANGUAGE PROCESSING, pp. 217-252.
5. <Bobrow 1969> Bobrow, Daniel, and J.B. Fraser, "An Augmented State Transition Network Analysis Procedure," Proc. of IJCAI, 1969, pp. 557-568.
6. <Borko 1967> Borko, Harold (ed.), AUTOMATED LANGUAGE PROCESSING, John Wiley and Sons, New York, 1967.
7. <Charniak 1969> Charniak, Eugene, "Computer Solution of Calculus Word Problems," Proc. of IJCAI, 1969, pp. 303-316.
8. <Chomsky 1957> Chomsky, Noam, SYNTACTIC STRUCTURES, Mouton and Co., The Hague, 1957.
9. <Chomsky 1965> Chomsky, Noam, ASPECTS OF THE THEORY OF SYNTAX, M.I.T. Press, Cambridge, Mass., 1965.
10. <Coles 1967> Coles, L. Stephen, "Syntax Directed Interpretation of Natural Language," Doctoral Dissertation, Carnegie Mellon University, 1967.
11. <Coles 1968> Coles, L. Stephen, "An On-Line Question-Answering System with Natural Language and Pictorial Input," Proc. National ACM Conference, 1968, pp. 157-167.
12. <Craig 1966> Craig, J.A., S.C. Berezner, H.C. Carney, and C.R. Longyear, "DEACON: Direct English Access and Control," Proc. FJCC 1966, pp. 365-380.
13. <Darlington 1964> Darlington, J., "Translating Ordinary Language Into Symbolic Logic," Memo MAC-M-149 Project MAC, M.I.T., 1964.

14. <Earley 1966> Earley, J.C., "Generating a Recognizer for a BNF Grammar," Comp Center Paper, Carnegie Mellon Univ., 1966.
15. <Feigenbaum 1963> Feigenbaum, Edward A., and J. Feldman, COMPUTERS AND THOUGHT, McGraw-Hill, New York, 1963.
16. <Fodor 1964> Fodor, J.A., and J.J. Katz, (ed.) THE STRUCTURE OF LANGUAGE, Prentice Hall, Englewood Cliffs, N.J., 1964.
17. <Fodor 1967> Fodor, J.A., and R. Garrett, "Some Syntactic Determinants of Sentential Complexity," PERCEPTION AND PSYCHOPHYSICS, 1967, Vol. 2(7).
18. <Garvin 1965> Garvin, P.L., et. al., "A Syntactic Analyzer Study -- Final Report," Bunker-Ramo Corp., Rome Air Development Center, RADC-TT-65-309, Dec. 1965.
19. <Green 1969a> Green, Cordell, "Application of Theorem Proving to Problem Solving," Proc. of IJCAI, 1969, pp. 219-240.
20. <Green 1969b> Green, Cordell, and B. Raphael, "The Use of Theorem-Proving Techniques in Question-Answering Systems," Proc. of ACM National Conference, 1968, pp. 169-181.
21. <Green, P. 1961> Green, P.F., A.K. Wolf, C. Chomsky, and K. Laugherty, "BASEBALL: An Automatic Question Answerer," in Feigenbaum and Feldman (ed.) COMPUTERS AND THOUGHT, pp. 207-216.
22. <Halliday 1961> Halliday, M.A.K., "Categories of the Theory of Grammar," WORD 17, 1961.
23. <Halliday 1966a> Halliday, M.A.K., "Some Notes on 'Deep' Grammar," JOURNAL OF LINGUISTICS 2, 1966.
24. <Halliday 1966b> Halliday, M.A.K., "The English Verbal Group: A Specimen of a Manual of Analysis" Nuffield Programme in Linguistics and English Teaching, Work Paper VI, 1966.
25. <Halliday 1967> Halliday, M.A.K., "Notes on Transitivity and Theme in English," JOURNAL OF LINGUISTICS 3, 1967.
26. <Halliday 1970> Halliday, M.A.K., "Functional Diversity in Language as Seen From a Consideration of Modality and Mood in English," FOUNDATIONS OF LANGUAGE 6 (1970), pp. 322-361.

27. <Hewitt 1969> Hewitt, Carl, "PLANNER: A Language for Proving Theorems in Robots," Proc. of IJCAI, 1969, pp. 295-301.
28. <Hewitt 1970> Hewitt, Carl, PLANNER, MAC-M-386, Project MAC, M.I.T. October, 1968, revised August, 1970.
29. <Huddleston 1965> Huddleston, R.D., "Rank and Depth," LANGUAGE 41, 1965.
30. <Hudson 1967> Hudson, R.A., "Constituency in a Systemic Description of the English Clause," LINGUA 17, 1967.
31. <Katz 1964> Katz, J.J., and J.A. Fodor, "The Structure of a Semantic Theory," In Fodor and Katz (ed.), THE STRUCTURE OF LANGUAGE, pp. 479-518.
32. <Kellogg 1968> Kellogg, C., "A Natural Language Compiler for On-line Data Management," Proc. of FJCC, 1968, pp. 473-492.
33. <Klima 1964> Klima, Edward S., "Negation in English," In Fodor and Katz (ed.), THE STRUCTURE OF LANGUAGE, pp. 246-323.
34. <Kuno 1965> Kuno, S. "The Predictive Analyzer and a Path Elimination Technique," CACM 8:7 (July 1965), pp. 453-462.
35. <Lindsay 1964> Lindsay, Robert, "Inferential Memory as the Basis of Machines Which Understand Natural Language," In Feigenbaum and Feldman (ed.) COMPUTERS AND THOUGHT, pp. 217-236.
36. <McConlogue 1965> McConlogue, K.L., and R. Simmons, "Analyzing English Syntax with a Pattern-Learning Parser," CACM 8:11 (November 1965), pp. 687-698.
37. <Michie 1968> Michie, D. (ed.), MACHINE INTELLIGENCE 3., American Elsevier Press, New York, 1968.
38. <Miller 1951> Miller, George, LANGUAGE AND COMMUNICATION, McGraw-Hill, New York, 1951.
39. <Minsky 1965> Minsky, Marvin, "Matter, Mind, and Models," In Minsky (ed.) SEMANTIC INFORMATION PROCESSING, pp. 425-432.
40. <Minsky 1968> Minsky, Marvin, (ed.) SEMANTIC INFORMATION PROCESSING, M.I.T. Press, Cambridge, Mass., 1968.

41. <Minsky 1970> Minsky, Marvin, "Form and Content in Computer Science," JACM, Jan., 1970.
42. <NAS 1966> National Academy of Sciences, LANGUAGE AND MACHINES: COMPUTERS IN TRANSLATION AND LINGUISTICS, National Academy of Sciences, Washington D.C., 1966.
43. <Petrick 1965> Petrick, S., "A Recognition Procedure for Transformational Grammars," Doctoral Dissertation, M.I.T., 1965.
44. <Quillian 1966> Quillian, M. Ross, "Semantic Memory," in Minsky (ed.) SEMANTIC INFORMATION PROCESSING, pp. 216-270.
45. <Quillian 1969> Quillian, M. Ross, "The Teachable Language Comprehender," CACM 12:8 (August 1969), pp. 459-475.
46. <Raphael 1964> Raphael, Bertram, "SIR: A Computer Program for Semantic Information Retrieval," in Minsky (ed.) SEMANTIC INFORMATION PROCESSING, pp. 33-134.
47. <Robinson 1965> Robinson, J.A., "A Machine-Oriented Logic Based on the Resolution Principle," JACM, 12:4 (October 1965), pp. 536-541.
48. <Shapiro 1969> Shapiro, Stuart C., and G.H. Woodmansee, "A Net Structure Based Relational Question Answerer: Description and Examples," Proc. of IJCAI, 1969, pp. 325-346.
49. <Siklossy 1968> Siklossy, L., "Natural Language Learning by Computer," Doctoral Dissertation, Carnegie Mellon Univ., 1968.
50. <Simmons 1966> Simmons, R.F., J.F. Burger, and R.E. Long, "An Approach Toward Answering English Questions from Text," Proc. of FJCC 1966, pp. 357-363.
51. <Simmons 1968> Simmons, R.F., J.F. Burger, and R. Schwarcz, "A Computational Model of Verbal Understanding," Proc. of FJCC, 1968, pp. 441-456.
52. <Slagle 1965> Slagle, James R., "Experiments with a Deductive Question-Answering Program," CACM, 8:12 (December 1965), pp. 792-798.
53. <Sussman 1970> Sussman, Gerald, T. Winograd, and E. Charniak, "Micro-Planner Reference Manual," AI Memo 203, Project MAC, M.I.T., July, 1970.

54. <Tharp 1969> Tharp, Alan L., and G.K. Krulee, "Using Relational Operators to Structure Long-Term Memory," Proc. of AJCAI, 1969, pp. 579-586.
55. <Thompson 1966> Thompson, F.B., "English for the Computer," Proc. of FJCC, 1966, pp. 349-356.
56. <Thorne 1968> Thorne, J., P. Bratley, and H. Dewar, "The Syntactic Analysis of English by Machine," In Michie, D. (ed.) MACHINE INTELLIGENCE 3., pp. 281-309.
57. <Thorne 1969> Thorne, J., "A Program for the Syntactic Analysis of English Sentences," CACM 12:8 (August 1969), pp. 475-480.
58. <Weizenbaum 1966> Weizenbaum, J., "ELIZA" CACM 9:1 (January 1966), pp. 36-45.
59. <Weizenbaum 1967> Weizenbaum, J., "Contextual Understanding by Computers," CACM 10:8 (August 1967), pp. 474-480.
60. <White 1970> White, Jon L., "Interim LISP Progress Report," AI Memo 190, Project MAC, M.I.T., March, 1970.
61. <Winograd 1968> Winograd, Terry, "Linguistics and the Computer Analysis of Tonal Harmony," JOURNAL OF MUSIC THEORY 12:1, 1968, pp. 2-49.
62. <Winograd 1969> Winograd, Terry, "An Interpretive Theory of Language", unpublished term paper, M.I.T., 1969.
63. <Woods 1967> Woods, William A., "Semantics for a Question-Answering System," Report No. NSF-19, Aiken Computation Laboratory, Harvard Univ., Sept., 1967.
64. <Woods 1968> Woods, W. "Procedural Semantics for a Question-Answer Machine," Proc. FJCC, 1968, pp. 457-471.
65. <Woods 1969> Woods, William A., "Augmented Transition Networks for Natural Language Analysis," Report No. CS-1, Aiken Computation Laboratory, Harvard Univ., Dec., 1969.
66. <Zwicky 1965> Zwicky, A.M., J. Friedman, B.C. Hall, and D.E. Walker, "The MITRE Syntactic Analysis Procedure for Transformational Grammars," Proc. of FJCC, 1965, pp. 317-326.

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R&D			
<i>(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)</i>			
1. ORIGINATING ACTIVITY (Corporate author) Massachusetts Institute of Technology Project MAC		2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	
		2b. GROUP None	
3. REPORT TITLE Procedures as a Representation for Data in a Computer Program for Understanding Natural Language			
4. DESCRIPTIVE NOTES (Type of report and inclusive dates) Ph.D. Thesis, Department of Mathematics, August 1970			
5. AUTHOR(S) (Last name, first name, initial) Winograd, Terry			
6. REPORT DATE February 1971		7a. TOTAL NO. OF PAGES 464	7b. NO. OF REFS 65
8a. CONTRACT OR GRANT NO. N00014-70-A-0362-0002		9. ORIGINATOR'S REPORT NUMBER(S) MAC TR-84 (THESIS)	
b. PROJECT NO.		9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)	
c.			
d.			
10. AVAILABILITY/LIMITATION NOTICES Distribution of this document is unlimited.			
11. SUPPLEMENTARY NOTES None		12. SPONSORING MILITARY ACTIVITY Advanced Research Projects Agency 3D-200 Pentagon Washington, D.C. 20301	
13. ABSTRACT This paper describes a system for the computer understanding of English. The system answers questions, executes commands, and accepts information in normal English dialog. It uses semantic information and context to understand discourse and to disambiguate sentences. It combines a complete syntactic analysis of each sentence with a "heuristic understander" which uses different kinds of information about a sentence, other parts of the discourse, and general information about the world in deciding what the sentence means.			
<div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 10px auto;"> DISTRIBUTION STATEMENT A Approved for public release; Distribution Unlimited </div>			
14. KEY WORDS			
Natural Language	Linguistics	Question-Answering	Machine Understanding
Theorem Proving	Semantics	Computer Discourse	Computational Linguistics
PLANNER	Parsing	English Grammar	Man-Machine Communication

DD FORM 1473 (M.I.T.)

NOV 68

UNCLASSIFIED

Security Classification

END
DATE
FILMED

5-5-71

UNCLASSIFIED
Technical
Report
distributed by



DEFENSE

TECHNICAL

INFORMATION

CENTER

DTIC

*Acquiring Information -
Imparting Knowledge*

DEFENSE LOGISTICS AGENCY
Cameron Station
Alexandria, Virginia 22304-6145

UNCLASSIFIED

UNCLASSIFIED

NOTICE

We are pleased to supply this document in response to your request.

The acquisition of technical reports, notes, memorandums, etc., is an active, ongoing program at the **Defense Technical Information Center (DTIC)** that depends, in part, on the efforts and interest of users and contributors.

Therefore, if you know of the existence of any significant reports, etc., that are not in the **DTIC** collection, we would appreciate receiving copies or information related to their sources and availability.

The appropriate regulations are Department of Defense Directive 3200.12, DoD Scientific and Technical Information Program; Department of Defense Directive 5200.20, Distribution Statements on Technical Documents (*amended by Secretary of Defense Memorandum, 18 Oct 1983, subject: Control of Unclassified Technology with Military Application*); Military Standard (*MIL-STD*) 847-B, Format Requirements for Scientific and Technical Reports Prepared by or for the Department of Defense; Department of Defense 5200.1R, Information Security Program Regulation.

Our Acquisition Section, **DTIC-FDAB**, will assist in resolving any questions you may have. Telephone numbers of that office are: (202) 274-6847, (202) 274-6874 or Autovon 284-6847, 284-6874.

DO NOT RETURN THIS DOCUMENT TO DTIC

**EACH ACTIVITY IS RESPONSIBLE FOR DESTRUCTION OF THIS
DOCUMENT ACCORDING TO APPLICABLE REGULATIONS.**

UNCLASSIFIED

PROPERTY OF DACS